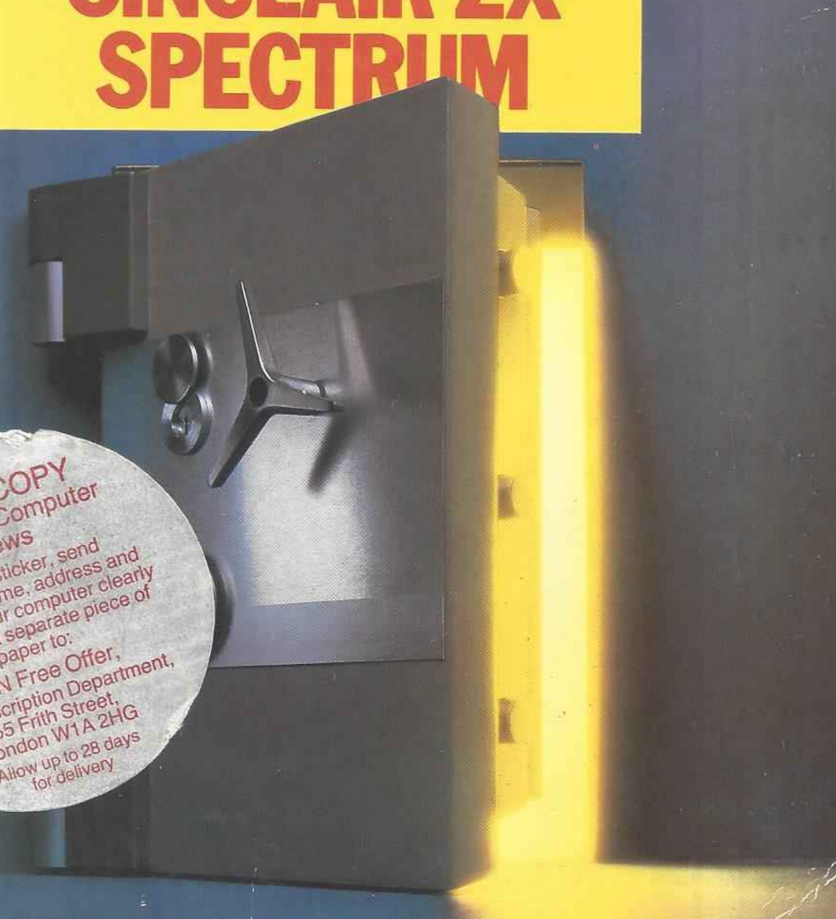
Pan

PERSONAL
COMPUTER
COMPUTER NEWS LIBRARY

JOHN WILSON

CRACKING THE CODE on the SINCLAIR ZX SPECTRUM



FREE COPY
Personal Computer
News
Peel off sticker, send
with your name, address and
details of your computer clearly
written on a separate piece of
paper to:
PCN Free Offer,
c/o Subscription Department,
55 Frith Street,
London W1A 2HG
Allow up to 28 days
for delivery

Pan/Personal Computer News
Computer Library

John Wilson

Cracking the code on the Sinclair ZX Spectrum

Pan Books London and Sydney

First published 1984 by Pan Books Ltd,
Cavaye Place, London SW10 9PG
in association with Personal Computer News
9 8 7 6 5 4 3 2 1

ISBN 0 330 28665 X
Photoset by Parker Typesetting Service, Leicester
Printed and bound in Great Britain by
Richard Clay (The Chaucer Press) Ltd, Bungay, Suffolk

Contents

Introduction

This book is intended for people with a reasonable knowledge of Sinclair BASIC and the Spectrum computer who now want to become proficient in machine code programming.

Machine code is like BASIC in that it is a language for communication with the computer, but it differs in the way that precise instructions have to be given in order to perform even the most simple of calculations and operations. These machine code instructions and their use will be introduced and explained with the aid of example programs, so that by the end of the book the reader and user (a lot depends on practice!) will be a competent machine code programmer.

The book starts by introducing the programmer to number representation and goes on to simple loading and storing techniques. It then proceeds through each set of instructions clearly and methodically, with plenty of examples.

After the explanation of the instruction set the use of a monitor is introduced and a full machine code monitor listing, which can be utilised to enter other machine code routines in this book, is provided. This is followed by a detailed breakdown of a machine code program.

Once these chapters have been digested the programmer can progress to dealing with more complex techniques. These involve using the ROM routines, screen handling, interrupts, and include a routine to handle sprites. Finally, we have a chapter which includes some useful machine code routines to enhance your own programs such as a sort, music, and pixel scroll routines. In the appendices are complete listings by Op code and Mnemonic of the Z80 instruction set.

1 Chips, registers and numbers

The Spectrum's central processing unit (CPU) or main control chip is known as the Z80. This powerful little chip handles all the additions, subtractions and logical operations with which the Spectrum implements your BASIC and machine code programs. To communicate to this chip when calculations and operations need to be done the user can of course type instructions in BASIC. However, there are other languages which can be used. The fastest of these is machine code which acts directly on the Z80 chip and can be very efficient. Machine code consists of a set of simple instructions which the Z80 CPU understands and can execute, such as addition, subtraction and comparison. This particular chip has over 700 instructions that can be sorted into a collection of a few different types. These instructions act upon data in the form of memory addresses and numbers.

BASIC is a very easy language in which to program due to the fact that we write a line of BASIC almost as we would say it in English, so that;

```
LET X=X+20*2+1
```

means set the variable X equal to the correct value of X, plus twenty times two, plus one. In machine code programming, however, we have to give more precise instructions at a low level, and specify each individual operation needed to perform the calculation.

The example above could be broken down to the sequence:

'add 20 to itself'	(2*20)
'add 1 to that resuit'	(2*20+1)
'add X to that result'	(X+2*20+1)

'and put the answer back in X'

It should be noted that the above is not an example of machine code instructions but simply illustrates the precision with which machine code operations have to be specified. Why should programmers use this complex sequence of machine code instructions when BASIC is

so easy? Let us look at an example to answer this question. First type in this BASIC program and RUN it:

```
10 FOR X=16384 TO 22527
20 POKE X,255
30 NEXT X
```

When RUN, the program very slowly fills the screen with ink. Now try running an equivalent machine code program:

```
  1 CLEAR 31999
 10 FOR x=32000 TO 32014
 20 READ a: POKE x,a
 30 NEXT x
 35 RANDOMIZE USR 32000
 40 DATA 33,0,64,1,0,24,54,255,
35,11,120,177,32,248,201
```

This program POKES a sequence of machine code instructions into the Spectrum RAM. The DATA at line 60 is the machine code program equivalent to the BASIC version given above. Each number represents a certain instruction which the computer's 'brain', the 'Z80 chip, executes. (Don't try to understand the code yet, just type it in!)

RUN the program . . . but don't blink, otherwise you will miss what happens! As you can see from the example, machine code is incredibly fast. An efficient machine code program can execute up to 1000 times as fast as the BASIC equivalent. What's more, machine code is also compact. You can write machine code routines which occupy only a quarter of the memory that their BASIC counterparts would.

Why is BASIC so slow? Well, the reason lies in the fact that the Z80 chip (which does all the calculations for the Spectrum) can only understand machine code. In order for it to execute a BASIC program, it first has to look up each BASIC keyword or token every time it reads a line. It then takes this token and translates or interprets it to specify the equivalent ROM machine code routine so that it can then perform the operation. This all takes time. Machine code, however, is the Z80's 'Mother Tongue', so no translation is needed and the code is executed immediately.

Hexadecimal and binary

All of you should know that the Spectrum (or any other computer for that matter) stores data in terms of 'bytes'. A byte is an 8 bit binary number which can have a decimal value of 0 to 255. In a 48K

Spectrum there are 49152 locations in memory where bytes can be stored. The value 49152 is obtained by the calculation 48*1024 because 1K=1024 bytes.

The Z80 chip stores numbers in groups of 8 bits, so it is known as an '8 bit chip'. In this it is similar to the 6502 chip which is used in the BBC Micro, Oric and Commodore machines. Other microprocessor chips use 16 or 32 bits and are therefore known as '16 bit' or '32 bit' chips.

To address RAM the Z80 chip uses 2 bytes (or 16 bits) This means that it can access 65536 characters, since the number of combinations of 16 1's and 0's is 65536. These bits and how they represent numbers and characters are best explained by looking at the system known as the *binary system* (or 'base two system').

In the real world of handling money we count in a system known as *decimal* or 'base 10 system'. We have the digits 0,1,2,3,4,5,6,7,8 and 9 which we can write to represent certain quantities.

In the decimal system we can break down the number we are using into groups of powers of ten. That is units, tens, hundreds, thousands, ten thousands, and soon. For example, the number 3456 can be broken down to:

3*1000	(3*10 ↑ 3)
+4*100	(4*10 ↑ 2)
+5*10	(5*10 ↑ 1)
+6*1	(6*10 ↑ 0)

In the binary system we use only two digits, these being 0 and 1. In order to represent large numbers therefore we can only write in a series of these two digits.

Remember that the Z80 chip represents information (numbers) in groups of 8 bits. Each of these bits may be 'off' (i.e. digit 0) or 'on' (i.e. digit 1). The bits in a byte are numbered 0 to 7, starting from the right.

In the binary system numbers are broken down in powers of two (that's why it is also known as the base two system). That is to say we break them down as factors of units (bit 0), two's (bit 1), four's (bit 2), eight (bit 3), sixteen (bit 4), thirty-two (bit 5), sixty-four (bit 6) and one hundred and twenty eight (bit 7).

Take for example the binary number 00011001, this represents the decimal number:

0*128	(0*2 ↑ 7)
+0*64	(0*2 ↑ 6)
+0*32	(0*2 ↑ 5)
+1*16	(1*2 ↑ 4)
+1*8	(1*2 ↑ 3)
+0*4	(0*2 ↑ 2)
+0*2	(0*2 ↑ 1)
+1*1	(1*2 ↑ 0)
25 decimal	

The maximum number that can be represented in 8 bit (one byte) binary form is therefore 11111111, which represents 255 in decimal (128+64+32+16+8+4+2+1).

In order to deal with larger numbers the Z80 has some 16 bit instructions. All memory addressing is done with 16 bits, so the total number of individual bytes that can be pointed to in memory (*addressed*) should be equal to the total number of combinations of a 16 digit binary number. This will be equal to the maximum value +1 (since the value zero is a unique combination).

To obtain the maximum value possible in a 16 digit binary number we must evaluate 1111111111111111. This has a value of:

1*32768	(1*2 ↑ 15)
+1*16384	(1*2 ↑ 14)
+1*8192	(1*2 ↑ 13)
+1*4096	(1*2 ↑ 12)
+1*2048	(1*2 ↑ 11)
+1*1024	(1*2 ↑ 10)
+1*512	(1*2 ↑ 9)
+1*256	(1*2 ↑ 8)
+1*128	(1*2 ↑ 7)
+1*64	(1*2 ↑ 6)
+1*32	(2*2 ↑ 5)
+1*16	(1*2 ↑ 4)
+1*8	(1*2 ↑ 3)
+1*4	(1*2 ↑ 2)
+1*2	(1*2 ↑ 1)
+1*1	(1*2 ↑ 0)
+1	
65536	
OR 64K (1K=1024 bytes)	

The Spectrum uses 16K of this for its BASIC ROM, which is why the maximum amount of RAM in a standard Spectrum is 48K.

When 16 bit values are stored in memory, something strange happens. Since 16 bit values are made from two bytes and only one byte can be held in one memory location, it follows that a 16 bit value must occupy two bytes in memory. The way that they are stored is that the least significant byte (LSB), which is the right hand group of 8 bits, is stored in the first address and the most significant byte (MSB), which is the left hand group of 8 bits, is stored in the next address. It would appear that this was a strange way for the chip designers to build the Z80 but the reason is that the Z80, like other common microprocessor chips, has evolved from simpler chips that did not have any 16 bit operations. These older chips only used 8 bits to address memory and so they could only address a maximum of 256 bytes (1/4 K). When the newer chips were designed the extra work involved in storing 16 bit values for addresses etc. was simplified. This was done by storing the old 8 bit address (the LSB) followed by the rest of the new 16 bit address. This does cause some problems for novice machine code programmers but soon you will understand. The following diagram should simplify the explanation.

```
16 bit value    1010001010110101
                <MSB> <LSB>

Address                               10110101
Address+1                             10100010
```

Address and Address+1 can be any two addresses in RAM.

Negative integer numbers

We mentioned earlier how we represent numbers on the Z80 by having 8 binary bits to represent positive numbers from 0 to 255 i.e. 00000000 binary to 11111111 binary. To represent negative numbers we can use a convention known as *signed integer representation*. Signed integer representation uses the most significant (or leftmost) bit of an integer to represent the sign. If the sign bit is 1 (high or set) then the number is negative, and if it is 0 (low or reset) then the number is positive. To get an 8 bit negative number binary representation we subtract the equivalent positive number value from 256

So, for example, the negative number −12 is equivalent to the number 256−12=244 decimal or 11110100 binary. Using signed integer representation we can represent numbers from −128 to +127 decimal. The Z80 chip, whether adding signed or normal

integers will deal automatically with any addition or subtraction.

The same applies for obtaining negative 16 bit values, with the exception that bit 15 will be set to 1 if the value is negative (instead of bit 7 as in 8 bit values). To get the 16 bit representation of −12 do the following: 65536−12=65524 decimal or 11111111 11110100. 16 bit values can be between −32768 to +32767.

A quick way of finding the negative representation of an 8 bit or 16 bit integer is to use a method known as *two's complement*. We first get the binary representation of the positive number and complement each of it's 8 or 16 bits and then add one to our new result. Complementing means that we transform each 0 into a 1 and each 1 is transformed into a 0. For example, suppose we wanted to find the binary representation of the number −180 decimal. The 16 bit binary pattern for the number 180 is 0000000010110100. The complement of this number is:

$$\begin{array}{lr} & 1111111101001011 \\ \text{adding one} \quad + & 1 \\ \hline & 1111111101001100 \end{array}$$

Since −180 is outside the range of −128 to +127 this value could not be held in a single byte.

Another number system we need to know before we go any further is the base 16 or *hexadecimal system*. Base 16 refers to the fact that this number system has 16 digits:

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14 and 15.

In order not to confuse between the number 10 and the digits 1,0 we write the five highest digits as:

A for the number 10
B for the number 11
C for the number 12
D for the number 13
E for the number 14
F for the number 15

So the sequence of digits becomes:

0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Using the same principle as found in the base two and base 10 systems we break down the number in a hexadecimal system into multiples of 16. Take, for example, the hex number 20 (H or h after a

number distinguishes between hex and decimal numbers) so that 20H is clearly different from 20 decimal. This means 2*16+0*1=32+0=32 decimal.

The number AFH means 16*A+F*1. Since A in hexadecimal notation is 10 decimal, and F is 15, this gives 16*10+15*1=160+15=175 decimal.

The hexadecimal notation is widely used by machine code programmers since it makes numbers easier to remember than binary but more significant than decimal. Because the hexadecimal system is based on 16 (10H), and 16 is 10000 in binary, there is a close relationship between the binary system that the Z80 chip uses and the hexadecimal notation that most programmers use. Unfortunately there is no simple relationship between the decimal and binary systems, as the table below should illustrate:

BINARY	DECIMAL	HEXADECIMAL
10101011	171	AB
00010010	18	12
10000001	129	81
11110000	240	F0

8 bit hexadecimal values have up to two digits. These each represent the value in one *nybble* of the byte. A nybble consists of four bits, either the leftmost four or the rightmost four and by taking the value of each nybble the Hexadecimal digit can be calculated. In the example above the binary value 10101011 is shown to have a hexadecimal value of AB. This can be illustrated by taking the high nybble (1010) which equals 10 decimal (A in hex) and the low nybble (1011) which equals 11 decimal (B in hex), then combining them in the same order to give ABH.

So far we have only seen machine code entered by POKEing numbers into memory. This method of writing machine code is tedious and makes it difficult to understand and debug the code, so the designers of the Z80 chip developed a standard set of *mnemonics* in which to write Z80 code.

These Mnemonics are English-like words which (hopefully!) signify the action a particular instruction performs. For example, the mnemonic RET means RETurn and is equivalent to the RETURN instruction in BASIC, ie. it tells the processor to continue with the main program after a subroutine was called.

In order to translate these mnemonics into data which the computer understands we will need to assemble them. This can be done by hand but more often by a utility known as an *assembler*. The

programmer first of all types in a program in standard mnemonics and then the program assembles these instructions into machine language. Most machine code programmers write assembly code and use an assembler to create their machine code

When an assembler translates the RET instruction it puts into memory the value for that instruction, which is 201 decimal, C9 hexadecimal or 11001001 in binary.

There are plenty of good assemblers for the Spectrum on the market ranging in price from around £7 to £14. Most of these will work on both the 16K and 48K models. The 'Devpac' package from Hi-Soft, as an example, is at the top of this price range, but is good value. In addition to the assembler it comes with another package known as a *monitor*. Alternatively, Chapter 7 provides you with your own monitor program for only the cost of wear and tear on the fingertips. This is a utility which will allow you to enter and experiment with the routines in this book.

A monitor program allows the machine code programmer to input and look at a program in hexadecimal form. Other features often included with it are utilities to set break points, look at the values held in the registers (the Z80 'variables') and to move, save and load blocks of memory. Both the Spectrum monitor provided in this book and the Devpac monitor have all these standard features. Devpac's also includes the capacity to move a single step at a time through a machine code program. There is also a *disassembler* in the Devpac package. This is a routine which is the opposite of an assembler for it converts machine code binary data into Z80 mnemonics.

When seeking an assembler for your Spectrum you are advised to buy one which allows you to assemble a program at different addresses in memory. Most assemblers have a command ORG (ORIGin) which tells the assembler the start address from which to assemble the program. This is illustrated in the assembler listings included in this book.

There are certain features of assembler listings that need to be explained here otherwise confusion may occur. Assemblers have a feature which enables them to use what are known as *'pseudo'* operators. These are used to place strings or numbers in memory and are not standard Z80 mnemonics. They are only a feature used in certain assemblers, including the one used for the listings in this book.

***DEFB** Define Byte*

Can sometimes be abbreviated to 'DB'. This places the following data in memory. For example:

DB 02H,04H

would place the number 2 followed by a 4 at the location where it is being assembled.

DEFW *Define Word*
This is similar to DEFB but is used to place a two byte number in memory. The low byte of the given number is placed in location where it is assembled. The high byte will follow, as we explained earlier when 16 bit values were introduced:

DEFW 7 (equivalent to DEFW 0007H)

is the same as:

DB 0,7 (equivalent to DB 00H,07H)

DEFS *Define Space*
The number following this Psuedo operator is the number of bytes which we want to reserve. So the operator:

DEFS 100

Would reserve 100 bytes.

EQU *Equate*
This instruction is used to give values to labels. The format is a label, followed by the EQU, followed by a number:

PLOT EQU 22E5H

The above would give the label PLOT the value 22E5 hex.

; Comment
In most assemblers the ; is used in the same manner as the BASIC REM to indicate a useful remark or comment. This is very useful because without helpful comments assemble code is harder to understand than BASIC because the operations are less immediately obvious.

Another feature of machine code assemblers is the facility to refer to memory addresses by means of labels. Instead of entering an instruction which says 'Jump to Address 31000', we can set a label at the address 31000. We could assign the label the name 'Fred', for example, and then give an instruction 'Jump to Address Fred'. This can greatly simplify our program structure and also enables meaningful label names to be assigned to sections of code.

If you use the appendices of this book you will be able to assemble your own machine code programs. The first thing you need to do is to write the assembly code (Mnemonics) for your program. I have provided an example below which will go into the printer buffer to avoid you having to CLEAR high memory space:

```
ORG 23296   ; Start the code at the printer buffer
LD HL,4000H
LD DE,4001H
LD BC,17FFH
LD (HL),0
LDIR
RET
```

The effect of this program is to remove all the ink from the screen How it does so is not important currently because it is serving only to demonstrate how you can get machine code to work without buying an assembler program.

The ORG is not a part of the machine code but it shows where in memory the machine code must be stored. This is the address into which we will start to POKE the data.

To obtain the data for each of the mnemonics above you will need to look them up in Appendix 2. As an example, the entry for LD HL, 4000H will read:

Mnemonic	Decimal	Hex
LD HL,XXXX	33 XXX XXX	21 XX XX

In order to get the hex for LD HL,4000H the 4000H must be converted into two bytes and reversed in order (due to the LSB/MSB storage convention explained earlier).

So, LD HL,4000H will assemble to 21 00 40 in hex or 33 0 64 in decimal. Since we will be using a BASIC program to POKE the code you will need to calculate the decimal values to be placed in the data statement. I have calculated the example for you but try to follow through the procedure to make sure you understand the principles involved.

HEX	DEC	MNEMONICS
		ORG 23296
21 00 40	33 0 64	LD HL,4000H
11 01 40	17 1 64	LD DE,4001H
01 FF 17	1 255 23	LDBC,17FFH
36 00	54 0	LD (HL),0
ED B0	237 176	LDIR
D9	201	RET

Now to enter this machine code program the following BASIC program could be used:

```
10 FOR I=0 TO 703:PRINT CHR$(32+INT (128*RND));:NEXT I
20 LET A=23296
30 READ B:IF B=-1 THEN GOTO 50
40 POKE A,B:LET A=A+1:GOTO 30
50 PRINT #0: "PRESS A KEY TO CLEAR":PAUSE 1:PAUSE 0
60 RANDOMIZE USR 23296
70 DATA 33,0,64,17,1,64,1,255,23,54,0,237,176,201,-1
```

As you can probably see, this would be a reasonable way to write small programs of up to about 100 bytes but to write your first full machine code 48K mega-game you will need an assembler to shorten the development time. Another considerable advantage of using an assembler program is that you can save the source code (assembly code or mnemonics). It than can be loaded back from tape or microdrive and errors can be corrected in the machine or object code.

Registers

The Z80 CPU has several registers available to the programmer. These can be used to hold numeric values similar to BASIC variables but the programmer is limited to 22 registers. Some of these registers can be used in pairs to hold 16 bit values. The older chips such as the 6502 are unable to do this. The Z80 registers are referenced by the names:

A,B,C,D,E,F,H,L,IX,IY,SP,PC,I and R

From these A, B, C, D, E, H, L can all hold 8 bit values and IX, IY, SP and PC will hold 16 bit values. Registers SP, PC, F, I and R have specific functions which will be explained later and are not used for holding user data. In addition to these there is a second set of A, B, C, D, E, F, H

and L registers which are usually referred to as A′, B′, C′, D′, E′, F′, H′ and L′. These two sets of registers cannot be used at the same time, so in order to access the alternate set a special instruction has to be used, 'EXX' (Exchange) which flips from one register set to the other. (Two exceptions here are the A′ and F′ registers which are exchanged using EX AF, AF′)

As mentioned earlier, some of these 8 bit registers can be paired off to form one sixteen bit register. The diagram below demonstrates how this can be done:

A	F
B	C
D	E
H	L

A′	F′
B′	C′
D′	E′
H′	L′

IX
IY
SP
PC

Now let's take a more detailed look at each of the registers that we have just introduced and their functions.

__IX__ and __IY__ Registers

These are known as the Index registers. The IX and IY registers are often used to point to tables of data and are extermely powerful tools for accessing arrays of data by a method known as indexing. On the Spectrum great care must be taken before using the IY index register in your own machine code programs. A number of ROM routines require that IY contains the value 5C3AH (23610 decimal) otherwise they will not work correctly. The Interrupt routine also requires this value to be in IY. Therefore if you must use IY in your machine code, disable the interrupts and make sure that IY=5C3AH before calling any ROM subroutines or returning to BASIC. Disabling interrupts and ROM subroutines are dealt with later in this book.

__I__ and __R__ Registers

The I or Interrupt register is used in conjunction with a technique known as *vectored interrupt programming*. This is the Z80's pointer

for alternative interrupt routines and is described in detail in Chapter 11.

The R or Refresh register is used to refresh any dynamic ram connected to the Z80. The only purpose it serves for Spectrum programmers is its use in random number generation.

A Register
The A register is known as the Accumulator and is the main register for performing 8 bit arithmetic and logical operations.

F Register
The F or Flag register indicates the state of certain arithmetic conditions after particular groups of instructions have been executed. A large number of Z80 instructions set flags depending on the values in various registers (usually A). When a flag is set, a bit in the F register is set to 1. For example if the result of a subtraction was zero the Z flag would be set to 1. That is, bit 6 of the F register would be on. There are other instructions that will only work if a particular flag is set. One example of this would be RET Z. This means if the Z flag is set then RET (return from subroutine), otherwise do nothing.

7	6	5	4	3	2	1	0
S	Z	X	H	X	P/V	N	C

The Flag register has 8 bits which can be either high or low (1 or 0). Each of these bits is set if certain conditions exist, although bits 3 and 5 are not actually used. If you want to see the mnemonics for each instruction and how the flags are affected you can find them in Appendix 3.

***Carry** flag*
The Carry flag indicates whether there was an overflow from bit 7 of a register. It is mostly affected by addition, subtraction or shift instructions. By overflow we mean that, for example, adding 250 to 250 would give a value of 500. However the maximum value that can be held in 8 bits is 255 so the actual value left would be 244. Since the Carry flag would be set we know that the real value is 244+256 (500). The same applies to 16 bit values where a result would exceed 65535.

Some sample instructions that use the result of this flag are:

RET C; RETURN IF CARRY FLAG SET
JP NC, ADDRESS; JUMP to Address if CARRY NOT SET

N flag
The N flag, know as the add/subtract flag, cannot be used directly by

the programmer. It is used by the Z80 chip to record whether the last operation was a subtraction or an addition.

Parity/overflow *flag*
This is a dual purpose flag. When used to indicate parity the Parity flag is set (i.e. 1) if there is an even number of bits in the byte set to one. It is reset (i.e. 0) if the number of bits set to one is odd.

The flag can be used to represent overflow, if it is set when an arithmetic overflow occurs during an arithmetic operation. This might happen in an addition or subtraction operation involving two numbers with the same sign (i.e. both positive or both negative) and it changes the sign in the result.

H *flag*
The Half carry flag is used to indicate a carry from bit 3 of a byte to bit 4 of a byte.

The H and N flags are used by the CPU in order to do something known as binary coded decimal arithmetic (more about this later!)

Zero *flag*
The Zero flag is set by certain instructions when the result of that execution is zero.

Sign *flag*
The Sign flag is set by certain instructions which show the sign of a result i.e. if the result was negative then the Sign flag would be set. If the result was positive then the Sign flag would be reset.

HL' *Register pair*
These are the alternate H and L registers working as a 16 bit HL' register. It is included here just to serve as a warning about using HL' in USR subroutines. HL' is used to point to the calculator stack during USR subroutines and BASIC will probably crash if you RETURN to BASIC with HL' changed.

PC *Register*
The PC, or Program Counter is a 16 bit register that holds the address in memory of the instruction currently being executed.

The ***SP*** *Register*
The SP or Stack Pointer is another 16 bit register. This one points to the current address at the top of the stack. Unlike the term *queue*, which indicates that literally the first item in is the first item out, the *stack* is a term used to represent data held in the reverse order, in which the last item placed will be the first item out. (This is sometimes known as a LIFO 'Last In First Out' list).

Imagine a pile of books onto which more books are placed. In order to get to the bottom of the pile the last book placed on top will have to be the first one removed. This analogy is very similar to the way in which the stack works on the Z80.

If we wish to call a routine in machine code we use an instruction CALL (This is similar to the GOSUB instruction in BASIC). When the Z80 executes a CALL instruction it places the return address onto the stack. The return address is always PC+3, because the CALL instruction is three bytes long and the subroutine must RETurn at the start of the next instruction after the CALL. It then gets the CALL address and puts this into the Program Counter (PC register). You will need to remember that the Program Counter points to the location of the instruction currently being executed, so the program will carry on running from that address. When the Z80 meets a RET instruction (RETurn) the chip then POPs the return address from the stack and places it back into the PC register.

This is very similar to what happens in a BASIC program when it executes the GOSUB command and then RETURNS. As well as saving return addresses, the stack can also be used to save data. (This can prove useful when you start to run out of registers.) For example we can save the HL register pair by using the instruction:

```
PUSH HL
```

This means 'PUSH the HL register pair on the stack'. We could now use the register pair for other calculations if we wanted to, knowing that we have a copy on the stack. To retrieve data from the stack we use the instruction:

```
POP HL
```

This means 'POP the data on top of the stack into the HL register pair'. It is important, however, to note the order in which we PUSH and POP data. For example, if we use the instructions:

```
PUSH HL
PUSH BC
```

we must remember to POP the data in the reverse order to that in which we originally pushed them. So to place the data back into the same registers we would need to use the instructions:

```
POP BC
POP HL
```

If we popped the data from the stack with:

```
POP HL
POP BC
```

then it would become apparent that the register pairs had been changed over. This can be a useful way of moving data within the chip but care must be taken when using the stack. Problems will arise when a PUSH or POP instruction is missing because a RET could POP some data and RETurn to the wrong address. A large proportion of machine code 'crashes' are caused by programmers wrongly using the stack in this way. *Remember 'Last In First Out'*
Let us examine the following code:

```
LD HL,0
PUSH HL
RET
```

The first instruction tells the computer to load the HL register pair with the number 0. The second is the PUSH instruction which places the HL pair onto the stack and leaves the number 0 on the top of the stack. The last instruction is the RETurn instruction which retrieves the last 16 bit number on the stack and places it into the program counter. Since the top of the stack contains 0 the program will start to run from address 0000 — Bad news if you have not SAVEd your program!

Loading and storing

In order to manipulate information from one register to the other, from RAM to registers and vice-versa, we need to use what is known as loading operations. These operations can be used on both 8 bit and 16 bit registers and constitute the major part of the Z80 instruction set. So learn them well!

First let us look at a few 8 bit LOAD operations:

```
3E 16      LD    A,22
```

The above instruction means 'LOAD the A register with the value 22 decimal'. It does precisely what it says: it puts the value 22 into the A register. The two digits on the lefthand side of the operation are its hexadecimal equivalent, which are POKEd into memory or typed in using a monitor. (An assembler does automatically.) We can also LOAD other 8 bit registers with data.

Examples

```
06 16  LD  B,22   ;LOAD B register with 22 decimal
06 22  LD  B,22H  ;LOAD C register with 22 hex
2E 04  LD  L,4    ;LOAD L register with 4 decimal
0E 0C  LD  C,12   ;LOAD C register with 12 decimal
16 10  LD  D,10H  ;LOAD D register with 10 hex
1E FF  LD  E,255  ;LOAD E register with 255 decimal
26 56  LD  H,56H  ;LOAD H register with 56 hex
```

Here too the hexadecimal translation is given on the lefthand side of the mnemonic.

If you look at the first two examples, which LOAD the B register, you might notice something similar in their hexadecimal output. The first byte (06H) is the same in both instances. It is not a coincidence. The first byte of the instruction is known as the Op code and tells the computer which register we are dealing with. The second byte is the actual data which we are LOADing into the register. It is important to note that it is not possible to have an instruction such as:

```
LD    A,289
```

'LoaD A register with 289 decimal'

This is because the number 289 takes more than 8 bits to represent it. We can however LOAD register *pairs* with 16 bit numbers.

16 bit LOADS

As we have mentioned before the Z80 chip has the facility for pairing off registers, a feature which gives access to some powerful 16 bit commands.

Let us recap which registers can be paired off together:

AF	AF′
BC	BC′
DE	DE′
HL	HL′

You can see from the diagram that the registers (with the exception of the Accumulator and Flag registers) are paired off in alphabetical order. The IX, IY, SP and PC registers have not been included in the diagram as these are true 16 bit registers and are not split into two like the others.

Let us now take a look at some 16 bit LOAD operations.

```
21 00 40      LD HL,16384
```

This means 'LoaD the HL register pair with 16384 decimal' (4000H) If you look at the hex translation, this time there are 3 bytes to represent the instruction. The first is the Op code for 'LD HL' and the last two are the data. The low part of the data is the second byte and the high part is the third byte. (Remember that the Z80 stores 16 bit values in the opposite way to which you would write them!)

Other examples of 16 bit LOAD operations are given below. (HH is the high byte of a number in hex while LL is the low byte).

```
   01  LL  HH  LD  BC,HHLL
   11  LL  HH  LD  DE,HHLL
   31  LL  HH  LD  SP,HHLL
DD 21  LL  HH  LD  IX,HHLL
FD 21  LL  HH  LD  IY,HHLL
```

Loading from one register to another

As well as LOADing 8 bit and 16 bit numbers into registers it is also possible to transfer information from one register into another.

Consider these examples:

```
78  LD  A,B
79  LD  A,C
6B  LD  L,E
```

The first example reads 'LoaD the A register with the B register'. If, for example, we had the instructions:

```
06  02  LD  B,2   ;load B register with 2
```

and then added the following instruction:

```
78  LD  A,B
```

we would find that the A register would take the contents of the B register, thus ending up with the value 2.

The Z80 chip does not have 16 bit instructions such as:

```
LD  HL,DE  ;load HL pair with DE pair????
```

so in order to achieve the same effect it is necessary to use a couple of 8 bit transfers, like this:

```
62  LD  H,D ;load H register with D register
6B  LD  L,E ;load L register with E register
```

Easy, isn't it!

The only 16 bit register to register load operations allowed in Z80 code are the following which deal exclusively with the stack pointer.

```
F9       LD  SP,HL
DD  F9   LD  SP,IX
FD  F9   LD  SP,IY
```

The next mode of addressing data is very similar to the way in which the BASIC instruction PEEK and POKE work. We are going to look at examples which load and store from locations in RAM and ROM.

```
3A  00  40  LD  A,(16384)
```

The instruction above reads 'LoaD the A register with the contents of the address 16384 (4000H). You can think of it as being similar to the BASIC instruction:

```
LET  x=PEEK(16384)
```

The number at the location 16384 is put into the A register. We could also put the contents of the A register into RAM by the instruction:

```
32  LL  HH  LD (ADDRESS),A
```

If we used the following instructions:

```
3E FF    LD A,255
32 00 40 LD (16384),A
```

the first instruction would LoaD the A register with the value 255 and the second would put the value of this register into the address 16384.

The Accumulator is the only 8 bit register which allows us to do this kind of addressing. There are *no* instructions such as:

```
LD (16384)B,  ;load the address 16384
               with b?
```

One way to get over this problem would be to use the instruction:

```
78          LD A,B         ;let A register=B register
32 00 40 LD (16384),A      ; put A register in 16384
```

Sixteen bit addressing in this mode is quite extensive; here are some examples of the instructions allowed.

```
ED 4B LL HH LD  BC,(HHLL)
ED 5B LL HH LD  DE,(HHLL)
ED 6B LL HH LD  HL,(HHLL) ;most assemblers use the faster form
                           of this instruction which is 2A LL HH
DD 2A LL HH LD  IX,(HHLL)
FD 2A LL HH LD  IY,(HHLL)
ED 7B LL HH LD  SP,(HHLL)
```

These instructions are 16 bit load instructions so they read two bytes from a given address. We could use:

```
2A 53 SC LD HL,(23635)
```

which reads 'LoaD the HL register pair with the contents of address 23635(5C53H). This would take the contents of the address 23635 and place them in the L register (low byte first). Finally it would take the contents of 23635+1 (i.e. 23636) and place it in the H register.

It is also possible to save the contents of registers at a given address, as follows:

```
ED 63 00 40 LD (16384),HL ;most assemblers would use the more
                           efficient 22 00 40 form of this
                           instruction
```

'LoaD at the location 16384 the value in the HL register pair.'

This instruction will put the value of the L register at the address 16384 and then put the value of H at the address 16385.

```
21 AA 22 LD HL,22AAh  ;load HL with 22AA hex
22 00 40 LD (16384),HL
```

The two instructions above would load AA hex at location 16384 and 22 hex at the location 16385.

Now suppose we wanted to load a value into the A register from an address which we did not directly know. The address can be worked out from a calculation. We would address that value by a method known as *register indirect addressing*. Sounds complicated, doesn't it? Don't worry, it's all very easy. All this means is that instead of giving an address directly to load from we have that address pointed to by a register pair, as you will see.

```
7E LD A,(HL)
```

The instruction above reads: 'LoaD the A register with the contents pointed by the address in the HL register pair'. If HL contained 16384 then the contents of that address would be put in the A register.

It is also possible to save using register indirect addressing, as follows:

```
   77 LD (HL),A
   12 LD (DE),A
36 22 LD (HL),22h  ;load 22h at the address in HL
```

The last instruction here is unique to the HL register pair. It is one of the most important and powerful register pairs available on the Z80 chip.

Last, but by no means least, is the powerful *index addressing mode*. These use the IX and IY registers and are extremely useful in accessing arrays of data.

The index modes are in the form:

```
DD RR NN      LD r,(IX+nn)
FD RR NN      LD r,(IY+nn)
DD RR NN      LD (IX+nn),r
FD RR NN      LD (IY+nn),r
DD 36 NN dd   LD (IX+nn),d
FD 36 NN dd   LD (IY+nn),d
```

where RR depends upon the register being used and dd represents the data. r is any of the registers A,B,C,D,E,H,L. nn is an offset with the value of 0 to 127 & 0 to −128. This is derived from the signed binary value of the number, which is added to the value of the index register. The store or load is then done at the resultant address. d is a byte value which can be loaded and stored directly.

Consider the following:

```
DD 21 00 60   LD  IX,6000h
DD 4E 05      LD  C,(IX+05)
DD 36 00 03   LD  (IX+00),03
```

address	data
6000	00
6001	02
6002	04
6003	05
6005	06
6006	07

After executing the first line the IX register is pointing to the portion of RAM/ROM at the address 6000 hex. When the second instruction is executed, the offset value 05 is added to the value of the IX register, which equals 6005 hex, and the contents of this location are put into the C register. Thus, the C register will contain the value 06. Note that the address in the IX register is not changed in any way. After executing this instruction it merely accesses the contents of that address. The last instruction:

```
LD (IX+00),03
```

goes through the similar process of working the offset address which is 6000+0=6000 hex and this time stores the value 03 at that address. The IY register works in a similar way . . . but a word of warning! If you are using the IY register on the Spectrum be very careful when mixing machine code with BASIC, as the Spectrum uses the IY register to point to the system variables. The procedure, as explained earlier when you were introduced to the IY register, would have to be applied when using the IY register in your own programs.

2 Number crunching

So far we have looked at the way the Z80 stores data and how it can transfer values and control from one address to another. In this chapter we come to the actual number crunching instructions used in addition and subtraction. As was pointed out before, the main advantage of the Z80 chip over other 8 bit microprocessors is that it can handle 16 bit numbers directly, making addition and subtraction operations that much easier. To begin with let's take a look at the 8 bit arithmetic instructions.

The two simplest number crunching instructions are DEC, 'DECrement register' and INC, 'INCrement register'. These two instructions respectively subtract or add 1 to the value in a specified register. We are allowed to use single registers A,B,C,D,E,L and H, with these instructions, so the range of possible commands is:

```
DEC A        INC A
DEC B        INC B
DEC C        INC C
DEC D        INC D
DEC E        INC E
DEC H        INC H
DEC L        INC L
```

The Accumulator or A register is one of the main registers in the Z80 chip and allows 8 bit arithmetic operations which can work directly with other registers and numbers. To add to the A register a value held in another register we use the instruction:

```
ADD A,r
```

which means 'Add to the Accumulator the value in register r', where r can be any register of A,B,C,D,E,H or L.

If we wanted to add numbers directly to the Accumulator we could use the instruction:

```
ADD A,N
```

where N is any 8 bit number. So for example, ADD A,5 would add 5 to the Accumulator.

We can also use the ADD instruction in conjunction with something known as *indirect addressing*. The HL register pair contains an address where the actual number which we wish to add to the Accumulator is stored:

```
ADD A,(HL)
```

The above instruction actually performs the operation 'add to the Accumulator the contents of the location pointed to by the register pair HL'. Take for example the following code:

```
LD A,8
LD HL,6000H
ADD A,(HL)
```

We'll assume we have the following data stored in memory from address 6000 hex onwards:

Address	Contents
6000H	02
6001H	03
6002H	06
6003H	07

The first instruction would set the Accumulator to 8 decimal. The HL register pair is then set to point to the address 6000 hex. The final instruction then gets the value from the address at HL (i.e. 6000 hex) and adds it to the Accumulator. This leaves it with the value 10 decimal.

Pursuing the indirect method even further, it can also be used with indexing utilising the IX or IY registers.

Using the same data and starting at address 6000 hex let us run through the following example to demonstrate this:

```
LD A,0
LD IX,6000H
ADD A,(IX+0)
ADD A,(IX+3)
```

The first and second instructions are simple enough. These set the A register to zero and the IX register to the address 6000 hex.

ADD A,(IX+0)

The instruction above adds the index to the address in the IX register. This new address is then used to point to the data which we wish to use. Since our index is zero, the address calculated is 6000H+0H=6000H. Therefore the contents are taken from this address and added to the accumulator, leaving it with a value of 2 after the first ADD instruction. The second addition is similar but uses the index 3, which means that the data to be added is stored at the address 6000H+03H=6003H and has the value 07. When this is added to the Accumulator the final result will be 9.

Subtraction works on the same registers as the ADD instruction, the mnemonic being SUB. Again every operation is done on the A register but the actual format of the mnemonic is slightly different as it does not actually mention the A register. The operand follows the SUB instruction directly. For example, to add the B register to the A register we would write:

ADD A,B

but to subtract the B register from the Accumulator we would write:

SUB B

Not too confusing, hopefully!

ADD A,A	SUB A
ADD A,B	SUB B
ADD A,C	SUB C
ADD A,D	SUB D
ADD A,E	SUB E
ADD A,H	SUB H
ADD A,L	SUB L
ADD A,(HL)	SUB (HL)
ADD A, (IX+d)	SUB (IX+d)
ADD A, (IY+d)	SUB (IY+d)
ADD A,N	SUB N

It is useful to note that ADD A, A is a quick and efficient instruction for doubling the value in the A register. SUB A is a quick way of setting the A register to zero (it works nearly twice as fast as LD A,0 and only takes up one byte instead of two).

Using the Carry flag

There is another set of 8 bit arithmetic instructions which take into account the state of the Carry flag. These are known as the ADC (Add with Carry) and SBC (Subtract with Carry).

In the case of addition the ADC adds the state of the carry flag as well as the given register or data. So, for example, if the A register contained 5 and the Carry flag was high (i.e. set to 1), if we ran the instruction:

```
ADC A,2
```

the answer left in the A register would be 5+2+1=8. On the other hand, if the Carry flag were to be reset we would return with the answer 7 as with the normal addition.

When it comes to subtraction, we subtract the state of the Carry flag from the Accumulator. So, if we had 5 in the A register and the Carry flag was set, the instruction:

```
SBC A,3
```

would leave the answer in the Accumulator as 5−2−1=2

```
ADC A,A            SBC A
ADC A,B            SBC A,B
ADC A,C            SBC A,C
ADC A,D            SBC A,D
ADC A,E            SBC A,E
ADC A,H            SBC A,H
ADC A,L            SBC A,L
ADC A,(HL)         SBC A,(HL)
ADC A,(IX+d)       SBC A,(IX+d)
ADC A,(IY+d)       SBC A,(IY+d)
```

The 16 bit increment and decrement instructions work in exactly the same manner as their 8 bit equivalents, but on pairs as opposed to single registers. The instruction DEC BC subtracts 1 from the value held in the BC register pair, while the instruction INC DE adds 1 to the DE pair. Because we are dealing with 16 bit operations we also have the option to increment or decrement the IX, IY and SP registers.

```
INC BC        DEC BC
INC DE        DEC DE
INC HL        DEC HL
INC IX        DEC IX
INC IY        DEC  IY
INC SP        DEC SP
```

16 bit addition is quite versatile on the Z8Ø. It allows the user to add (with or without Carry) other 16 bit registers to the HL, IX or IY register pair. Subtraction, however, is limited to subtracting the registers BC, DE, HL and SP from the HL pair and we only have the use of the Subtract with Carry instruction.

```
ADD HL,BC     ADC HL,BC
ADD HL,DE     ADC HL,DE
ADD HL,HL     ADC HL,HL
ADD HL,SP     ADC HL,SP
ADD IX,BC     ADD IY,BC
ADD IX,DE     ADD IY,DE
ADD IX,IX     ADD IY,IY
ADD IX,SP     ADD IY,SP
```

16 Bit subtraction.

```
SBC HL,BC
SBC HL,DE
SBC HL,HL
SBC HL,SP
```

Let us look at a few examples using some of these instructions.

```
LD HL,Ø432H
LD BC,Ø536H
ADD HL,BC
```

The above instructions would result in the HL pair containing Ø432H+Ø536H=Ø968H.

The ADD HL,HL instruction has the same effect as multiplying by 2. Combined with additional instructions it could be used to multiply a number by a power of two. For example, suppose we wished to multiply the contents in the DE pair by 32. First we transfer DE into HL, then we do five ADD HL,HL instructions in order to multiply by 32, and finally we transfer the answer back into DE like this:

```
;multiply DE pair by 32
```

```
EX DE,HL      ;SWOP DE AND HL
ADD HL,HL     ;TIMES BY 2
ADD HL,HL     ;TIMES BY 4
ADD HL,HL     ;TIMES BY 8
ADD HL,HL     ;TIMES BY 16
ADD HL,HL     ;TIMES BY 32
EX DE,HL      ;SWOP DE AND HL
              ;ANSWER IS NOW IN DE
```

The first and last instructions EX DE,HL mean 'exchange the DE and HL registers'. What they actually do is simply to swop the contents of the DE pair for the contents of the HL pair.

As we mentioned earlier, the Add with Carry instruction ADC takes into account the state of the Carry flag. For example, if the Carry flag were set and we used the instruction:

```
LD HL,0432H
LD BC,0536H
ADC HL,BC
```

the HL pair would contain 0432H+0536H=0968H+1 (state of Carry)=0969H. It is worth repeating that the only form of subtraction available with the 16 bit set is using the SBC instruction which also subtracts the state of the Carry flag to give the final result. Therefore, it is sometimes necessary to clear or reset this Carry flag before executing an SBC instruction in order to obtain the correct result. The way to do this is very simple. We use the 1 byte instruction:

```
AND A
```

This means 'AND the Accumulator with itself'. This is known as a *logical operation*, a process which we will be looking at more closely in chapter 5. All you need to know for now is that one of the effects fo this instruction is to reset the Carry flag. Thus in order to subtract 0432 hex from 0563 hex we could use the following piece of code:

```
LD HL,0536H   ;Put first number in HL
LD DE,0432H   ;Put second number in DE
AND A         ;clear the carry flag
SBC HL,DE     ;do the subtraction!
```

This should leave the result 0536H–0432H–0 (state of Carry)=104 hex

If we had not used the AND A instruction as a precaution to clear the Carry flag and if the Carry flag was set after the execution of a previous instruction, the result would be 0536H–0432H–1 (state of Carry)=103H.

Jumping and calling

In Spectrum BASIC we transfer control from one part of a program to another using the BASIC instructions GOTO and GOSUB. In order to implement transfers in machine code we use the JUMP and CALL instructions.

The simplest of these instructions is the JUMP *to address* command:

```
C3 00 60        JP 6000H
```

The above example reads 'JUMP to the address 6000 hex' and it loads the program counter with 6000 hex from where it will continue to execute the machine code.

We can also specify the address to JUMP to by the register pairs HL,IX and IY. For example, if we had the instruction:

```
JP (HL)
```

This would in effect load the program counter with the HL register pair. So if the HL pair contained 1601 hex the program would JUMP to the address 1601 hex.

In order to implement the equivalent of the BASIC statement 'IF condition THEN GOTO' we have to use something known as conditional jump instructions. There are eight conditions which can be identified, all of which are indicated by bits set in the flags register (F-register). Below we give all the conditional jump statements that are allowed:

```
JP NO,address ;'Jump if Carry flag reset (Non Carry)'
              ;to the address specified
JP C,address  ;'JUMP if Carry flag set (Carry)'
JP NZ,address ;'JUMP if Zero flag reset (non Zero)'
JP Z,address  ;'JUMP if Zero flag set (Zero)'
JP P,address  ;'JUMP if positive (Sign flag reset)'
JP M,address  ;'JUMP if minus (Sign flag set)'
JP PO,address ;'JUMP if Parity odd (Parity reset)'
JP PE,address ;'JUMP if Parity even (Parity set)'
```

Jump *relative*

There is another range of JUMP instructions available on the Z80, known as the *JUMP relative command*. This instruction allows us to specify an offset instead of an absolute address. The offset is a one byte number and allows us to jump backwards by up to 128 bytes and forwards up to 127 bytes, counted from the first byte after the instruction. This is because by using signed integer representation (see chapter 1) a byte can hold values between +127 and −128. The actual instruction is written as follows:

```
28 dd          JR dd
```

JUMP relative dd bytes, where dd is the displacement to JUMP. For example, in the case below:

```
18 03          JR 03
00             NOP
00             NOP
00             NOP
3E 04          LD A,4
```

the code would load the JUMP past the two NOP (No operation) instructions to the instruction which LOADS the Accumulator with the value 4. The displacement 02 is added to the location after the JUMP instruction. Since the JUMP relative instruction is two bytes long the actual address to which the program is transferred is the address of the JUMP relative instruction *plus* the displacement *plus* 2:

new address=old address+displacement+2

If you are using a monitor to type in a machine code program you will have to work out the displacement for yourself. However, most Z80 assemblers will let you reference addresses as labels and will automaticcally work out the displacement needed. So you could write the code like this:

```
        JR Here
        NOP
        NOP
Here    LD A,4
```

When assembled the displacement would be placed with the appropriate value.

Like the absolute JUMP the relative JUMP also has conditional

options. However, these are limited to the testing of the carry and the zero flags:

```
JR C,dd     ;'JUMP relative on Carry (Carry flag set)'
JR NC,dd    ;'JUMP relative non Carry (Carry flag reset)'
JR Z,dd     ;'JUMP relative on Zero (Zero flag set)'
JR NZ,dd    ;'JUMP relative non Zero (Zero flag reset)'
```

The advantage of using the JUMP relative instructions as opposed to those of the JUMP absolute lies in relative addressing. This takes only two bytes as compared to the three needed for the absolute mode, making a routine smaller in size. It also allows some particular routines to be relocateable, that is, having the ability to be placed anywhere in memory without having to be re-assembled.

DJNZ

The DJNZ 'DECrement JUMP on non zero' is an extremely powerful instruction. It allows the programmer to effect a loop a specified number of times around a portion of code, very much like the 'FOR . . . NEXT' statements in BASIC. Take a look at the following machine code program:

```
        LD B,20H
        LD HL,5800H
        LD A,2
LOOP    LD (HL),A
        INC HL
        DJNZ LOOP
        RET
```

The first instruction LD B,20H LOADs the B register with the number 20 hex (32 decimal). The B register is used as a loop counter for DJNZ.

We then LOAD the HL register with the two byte number 5800 hex. This is the start of the attribute file:

```
LD HL,5800H
```

The Accumulator is LOADed with the value 2, the colour code for red INK, black PAPER, BRIGHT 0 and FLASH 0. The next three instructions form the main part of the loop:

```
LOOP    LD (HL),A
        INC HL
        DJNZ LOOP
```

The value in the Accumulator is placed at the address pointed by the HL pair. When executed the first time round, the loop will load the value 2 into the start of the attribute file. Next we have the instruction:

```
INC HL
```

This means 'increment the HL register pair by one' and adds one to the HL pair so that it points to the next address in the attribute file. Finally we have:

```
DJNZ LOOP
RET
```

The DJNZ instruction will subtract one from the B register. If the value after this subtraction is not zero then it will JUMP relative to the address specified. If it is zero then it will go on to the next instruction which is a RETurn.

As you can see, DJNZ is an extremely powerful instruction. It is very much like having two instructions in one – a subtraction on the B register and a JUMP relative on non zero.

Bearing in mind that the DJNZ instruction uses relative and not absolute addressing we can only use it if the portion of code we are looping around is no longer than 128 bytes.

Calling and returning

The second method of transferring the control of a program is by using the set of CALL instructions.

There are times when a program executes the same portion of code many times or when other portions of code closely resembling each other are run with different parameters. Instead of having these similar routines scattered around at various different places in memory, you could have just one copy of this code when necessary, call it as a subroutine, very much like setting up a subroutine in BASIC using the 'GOSUB' BASIC instruction.

You can call this piece of code by using the instruction CALL followed by an address. The flow of the program will transfer to this address after storing the address of the instruction following the CALL instruction. The program is then executed normally until it reaches a RET (return) instruction, when it returns to the next instruction after the address of the call.

The CALL instruction takes this syntax:

```
CD LL HH        CALL HHLL
```

HH is the high byte of the address and LL is the low byte. It is possible for CALLs to be *nested*, which means that one subroutine may CALL another subroutine. If fact the number of nested calls allowed is limited only by the amount of memory left to the programmer. A subroutine may also call itself, a function known as recursion which is too abstruse for us to pursue here in any depth.

Like JUMP, the CALL and RETurn instructions also have conditional counterparts. We can CALL a subroutine or RETurn from a subroutine depending on the conditions set in the Flags register:

```
CALL HHLL        RET
CALL Z,HHLL      RET Z
CALL NZ,HHLL     RET NZ
CALL C,HHLL      RET C
CALL NC,HHLL     RET NC
CALL PO,HHLL     RET PO
CALL PE,HHLL     RET PE
CALL M,HHLL      RET M
CALL P,HHLL      RET P
```

There is another range of calling instructions, known as the restart (RST) set. They differ from the others in that they are only one byte long and are limited to CALLing one of eight addresses: 00 hex, 08 hex, 10 hex, 18 hex, 20 hex, 28 hex, 30 hex and 38 hex.

As you have probably noticed, all these addresses are in the ROM memory map which you may not find much use as we cannot write any code there. Well that's true, but we can CALL some of the routines from our own programs. Below are the CALLs and the object of the particular routines.

```
RST 00H   ;start boot up
```

This is a bit like typing NEW in BASIC, so is not very useful unless you wish to return to BASIC from a machine code program and protect the routine from prying eyes.

```
RST 08H   ;error restart
```

This routine is used by BASIC to report error messages. The error number is the byte following the restart instruction. It will give the error report of the data plus one. Thus:

```
RST 08H
DB 08
```

will generate the error message 09 'STOP statement'.

```
RST 10H    ;print a character
```

This is an exteremely useful routine. It prints the character in the Accumulator to the current channel. A channel outputs to a 'device', which can be either the printer or various parts of the screen. We'll see more of this in Chapter 9. A simple example for now is:

```
LD A,66    ;print the character B to the
RST 10H    ;current channel
```

```
RST 28H    ;floating point calculator
```

The number crunching routine above allows us easily to implement complex floating point arithmetic routines in machine code using the ROM functions. The floating point calculator is explained in more detail in Chapter 9.

```
RST 30H    ;make space
```

This is not a particularly useful routine. It simply creates space in the workspace area.

Finally:

```
RST 38H    ;scan the keyboard
```

This routine updates the system variable LAST–K and can be used to ascertain which keys are depressed. It is called 50 times a second by BASIC. It is also sometimes known as the Mode 1 maskable interrupt routine. We'll be hearing more about this routine when we get to Chapter 11, which deals with interrupts and their uses.

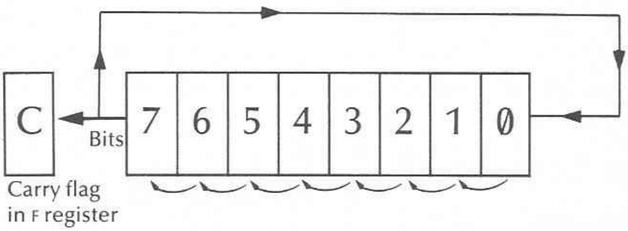
3 Rotating and shifting

Rotating and shifting operations provide the programmer with the means to manipulate the pattern of bits held in a register or a byte memory. These instructions, which are most useful for multiplication and division by powers of two, act on most of the 8 bit registers. They can use both indirect and index addressing modes. All the rotate instructions use the Carry flag (which is held in the F register) as a ninth bit, bit 8, therefore allowing the programmer to rotate this from the left or right through the register or memory. This should become clearer as we run through the available instructions.

Rotating

RLC *Rotate Left Circular*

This instruction rotates each bit of a given register or memory byte to the left by one bit. Bit 7 of the register or byte specified is rotated to the Carry flag and the same value is 'wrapped round' to bit 0:



For example, if the byte on which we were operating held 10101010 the following would occur after the RLC instruction was executed. The value of bit 7 (1) would be transferred to the Carry flag bit and to bit 0 of the byte with each of bits 0 to 6 shifted one place to the left. The result would be 01010101 stored in the byte, and the Carry flag set.

The RLC instruction can act on the registers A,B,C,D,E,H,L, as well as (HL) and (IY+INDEX) and (IX+INDEX). There is also a RLCA instruction which has the same effect as RLC A but is one byte shorter and twice

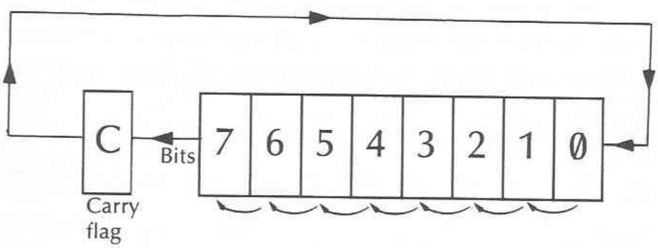
as fast to execute. These additional,short–form, rotate instructions on the Accumulator are available on all the rotate instructions. These are as follows (note that d indicates the index value where applicable):

RLCA	RLC (HL)
RLC A	RLC (IX+d)
RLC B	RLC (IY+d)
RLC C	
RLC D	
RLC E	
RLC H	
RLC L	

RL *Rotate left*

This instruction rotates the register left through all the nine bits, wrapping around the carry bit value to bit 0.

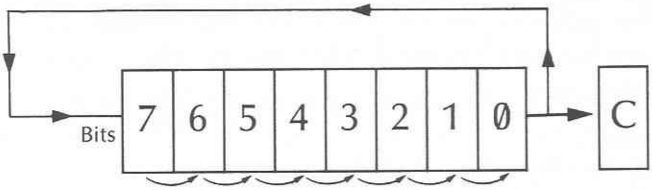
The effect of this instruction is to take the sequence of bits in the byte, add the Carry flag value as bit 8, and then shift all bits one place to the left. The Carry flag value then goes into bit 0. Thus if we had (1)01010101 before an RL instruction, we would end up with (0)10101011. This would produce a result which is the original value multiplied by two, *plus* the value of the Carry flag.



The available instructions are:

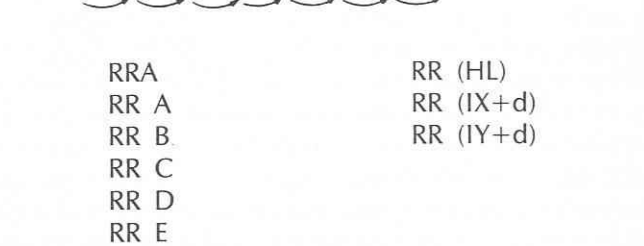
RLA	RL (HL)
RL A	RL (IX+d)
RL B	RL (IY+d)
RL C	
RL D	
RL E	
RL H	
RL L	

RRC *Rotate Right Circular*
The register or byte is rotated right from bit 7 through to bit 6 and so on. Bit 0 is then rotated to the Carry flag and bit 7. This is the reverse operation to that of RLC



RRCA	RRC (HL)
RRC A	RRC (IX+d)
RRC B	RRC (IY+d)
RRC C	
RRC D	
RRC E	
RRC H	
RRC L	

RR *Rotate Right*
The Rotate Right instruction has the opposite effect to that of the RL Rotate left instruction. Bit 0 of the register or byte is rotated to the right through the Carry, while the old Carry is rotated down to bit 7.

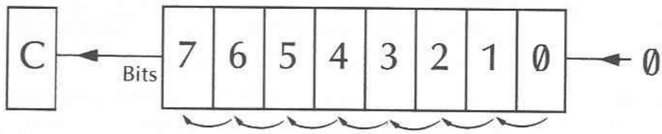


RRA	RR (HL)
RR A	RR (IX+d)
RR B	RR (IY+d)
RR C	
RR D	
RR E	
RR H	
RR L	

Shifting

As well as the Rotate instructions, there is also available a set of shift instructions which can make registers shift either left or right. This differs from the Rotate instruction set in that there is no 'wrap around' effect. Therefore one bit at either end of the byte is lost and a zero goes into this bit. Like the Rotate set all the shifts can act on A,B,C,D,E,H,L,(HL)(IX+d) and (IY+d).

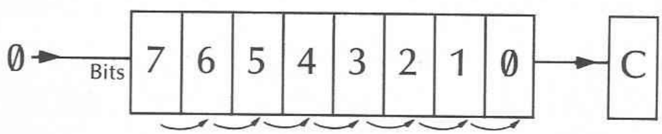
SLA *Shift Left Arithmetic*
The content of the carry bit is lost and the whole byte or register shifts to the left. Bit seven is shifted into the Carry flag, and a 0 inserted in bit 0.



SLA A
SLA B
SLA C
SLA D
SLA E
SLA H
SLA L

SLA (HL)
SLA (IX+d)
SLA (IY+d)

SRL *Shift Right Logically*
The SRL 'Shift Right Logically' shifts the bits from the left to the right, so is useful for dividing numbers by powers of two. Bit zero of the register/byte is shifted into the carry bit and a zero is placed into bit seven.

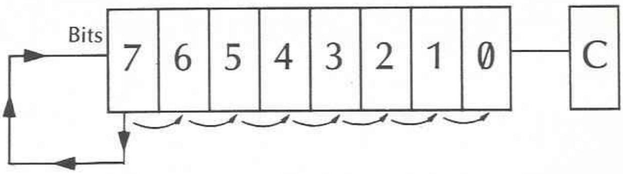


SRL A
SRL B
SRL C
SRL D
SRL E
SRL H
SRL L

SRL (HL)
SRL (IX+d)
SRL (IY+d)

***SRA** Shift Right Arithmetic*

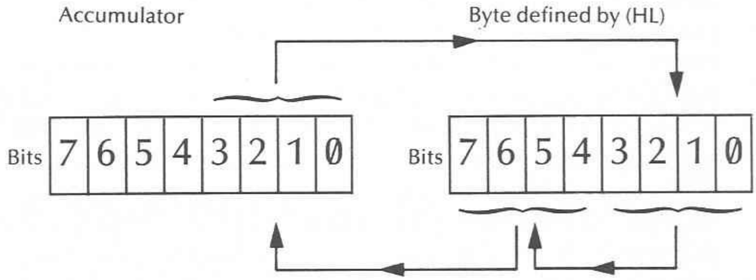
This is an odd instruction. Shift Right Arithmetic is identical to the SRL instruction apart from the fact that bit seven is left unchanged. This instruction is used to divide 'signed' numbers (i.e. numbers −127 to +128) by powers of two as it doesn't affect the sign bit.



SRA A	SRA (HL)
SRA B	SRA (IX+d)
SRA C	SRA (IY+d)
SRA D	
SRA E	
SRA H	
SRA L	

***RLD (HL)** ROTATE LEFT DECIMAL*

This is a single instruction which acts on both the accumulator and the contents pointed to by the HL register pair. It actually moves 'half bytes' called 'nybbles' from the Accumulator to a RAM location and vice versa.



As you can see from the diagram the bottom four bits (bits 3–0) of the location pointed by the HL register pair are shifted to the top four bits positions (7–4). The original top four bits are placed in the lower half of the accumulator with the original contents placed in the bottom four bits of the RAM location. If, for example, we had the HL pair containing **6000** hex, this byte holding CB hex, and the accumulator containing 2A:

Address	Contents	Accumulator
6000	2A	CB

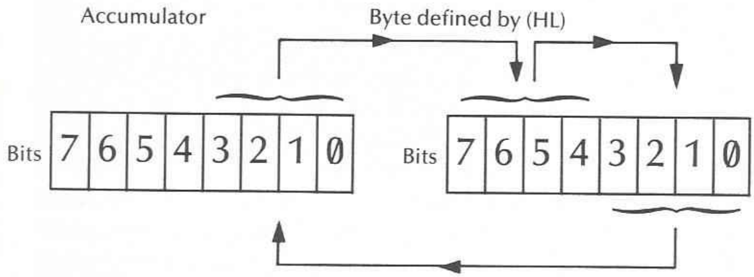
Then after executing the instruction

RLD (HL)

We would find that the contents of location 6000H and the accumulator would be changed to:

Address	Contents	Accumulator
6000	AB	C2

The instruction RRD (HL) has the opposite effect, as shown in the diagram below:



As already indicated, the shift instructions are very useful for multiplying and dividing by powers of two. If, for example, the Accumulator contained the value three and we had the instructions:

```
SLA A
SLA A
```

Then the result remaining in the Accumulator would be 12. Remember that the shift instructions affect the Carry flag, so if we had executed the instructions:

```
LD  A,128
SLA A
```

the bit pattern for 128 is 1000000. Therefore when the SLA A instruction is carried out, the top bit of the Accumulator would be shifted into the Carry flag. Zero remains in the A register leaving the Carry flag and Zero flag set. It is easy to write small routines to multiply

registers by numbers which are not multiples of two. For example to multiply a number by 10 simply split the calculation into two parts. First multiply the number by eight and then add twice the original number.

MULT10

```
SLA A    ;LET A=2*A
LD  B,A  ;LET B=A (2*original A)
SLA A    ;LET A=2*A (4*original A)
SLA A    ;LET A=2*A (8*original A)
ADD A,B  ;LET A=A+B (10*original A)
```

The first two instructions:

```
SLA A
LD B,A
```

multiply the Accumulator by two and save the result in the B register. Remember, the instruction

```
LD B,A
```

has no effect on the Accumulator but copies its contents into the B register. Therefore at this point we have double the original number in both the A and the B registers.

```
SLA A
SLA A
```

The two other shift instructions multiply the number by eight. Finally, the last instruction:

```
ADD A,B
```

adds the contents of the B register, which contains twice our original number, to the A register. This leaves the desired answer.

This method of multiplication would only work for numbers in the range of 0 to 25. Any larger number would result in a number greater than 255 which we are unable to fit into an eight bit byte. To perform multiplication on two byte numbers, using shifts, we have to take into account that a Carry may occur from the lower half of a register. This must be shifted to the high part. Therefore to multiply the HL register pair by two we use the instructions:

```
SLA L                    ;multiply lower part by two
RL H                     ;rotate putting carry into bit
                         ;0 in high register.
```

If we wanted to multiply the HL register pair by ten we could write:

```
SLA L
RL H         ;2*HL
LD E,L
LD D,H       ;Save in DE.ie DE=2*HL
SLA L
RL H         ;4*HL
SLA L
RL H         ;8*HL
ADD HL,DE ;HL=8*HL+2*HL=10*HL
```

Of course it would be much easier to use the ADD instruction to perform the multiplication.

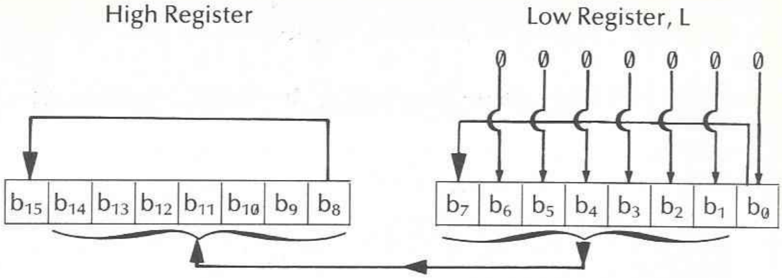
```
ADD HL,HL ;2*HL
LD E,L
LD D,H       ;DE=2*HL
ADD HL,HL ;4*HL
ADD HL,HL ;8*HL
ADD HL,DE ;10*HL
```

This piece of machine code is much faster and more concise to use than the previous example. However, this is not always the case. Suppose we wanted to multiply the HL pair by 128. The first thing that you thought of was probably to use the series of ADD instructions.

```
ADD HL,HL ;2*HL
ADD HL,HL ;4*HL
ADD HL,HL ;8*HL
ADD HL,HL ;16*HL
ADD HL,HL ;32*HL
ADD HL,HL ;64*HL
ADD HL,HL ;128*HL
```

A lot of instructions!

If we take a look at the bit pattern of a two byte number when we multiply be 128 we might be able to use shift and rotate instructions to our advantage. Let's look at the bit pattern we must get in order to multiply a two byte number by 128:



The top seven bits of the low byte need to be shifted into the bottom seven bits of the high byte. Bit 0 of the low byte will be shifted up to bit seven and bit seven of the high byte is lost.

If we represent the bit patterns by having hn representing bit n of the high byte and ln to represent bit n of the low byte then before we perform the multiplication we have the pattern:

h7	h6	h5	h4	h3	h2	h1	h0	high byte
l7	l6	l5	l4	l3	l2	l1	l0	low byte

After multiplying a two byte number by 128 we end up with the bit pattern:

h0	l7	l6	l5	l4	l3	l2	l1	high byte
l0	0	0	0	0	0	0	0	low byte

Notice that the first seven bits of the low byte will always be set to zero. Looking at the pattern we can see that we can get the new high byte pattern by shifting the old low byte to the left one. Before we do this we can put h0 into the Carry flag using the instruction:

```
SRL H
```

Now we have bit 0 of the H register i.e. h0 in the Carry flag. We can now get the pattern we need for our new high byte in the low byte L register by using the instruction.

```
RR L
```

This puts the Carry (containing the old value h0) into bit seven of the low byte. All the other bits are shifted to the right forcing the carry into the topmost bit. We now have the pattern we want for the H

register in the L register, so we transfer this by a simple LOAD command:

```
LD H,L
```

Finally, we set bit seven of the low byte to the contents of the Carry which is the bit l0. We do this by first setting the L register to zero and then rotating the Carry through to bit seven:

```
LD L,0
RR L
```

So our code for multiplying the HL register pair by 128 looks like this:

```
MULT128:    SRL H
            RR L
            LD H,L
            LD L,0
            RR L
```

The code is much smaller and faster to use than the series of ADD instructions. This portion of code is as much as 50% faster than the equivalent shown earlier. Arithmetic using bit manipulation is a little difficult to grasp at first but its implications are enormous. Screen addresses, for example, can be calculated much faster, giving games of infinite quality. Therefore it's very worthwhile to take some time to learn. Meanwhile, I'll end the chapter by giving you a routine to divide the HL register pair by 128 and let you find out how it works.

```
DIV128:     SLA L
            RL H
            LD L,H
            LD H,0
            RL H
```

4 Making comparisons and checking bits

The Compare Instruction

A COMPARE instruction operates in a similar fashion to a subtraction operation, except that the Accumulator is not changed. Instead, various flags are set or reset according to the result.

This instruction is most useful when used in conjunction with the Z80's conditional CALL and JUMP instructions and can be used to implement machine code equivalents of such BASIC statements as:

```
IF X>N  THEN GOTO ADDR
IF X<N  THEN GOTO ADDR
IF X=0  THEN GOTO ADDR
IF X<>0 THEN GOTO ADDR
```

The COMPARE instruction is limited to comparisons between eight bit numbers specified directly, indirectly, or contained in registers. Suppose we wanted to COMPARE the current value held in the Accumulator with the number 128 decimal.

We would use the instruction:

```
CP 128
```

which reads 'ComPare the current Accumulator value with the number 128'. This will set the Zero flag if the number in the Accumulator is equal to 128. The Carry flag will be set and the Zero flag reset if the number is greater than 128. Both the Carry flag and Zero flag would be reset if it was less than 128.

The following routine is a good example of the use of the COMPARE instruction with conditional branches to simulate the BASIC language IF . . . THEN . . . ELSE structure. The routine compares the A register with the B register and branches off to certain addresses, depending on whether the A and B registers are found to be equal, greater than, or less than A.

```
        CP    B
        JR    Z,EQUAL
        JR    C,BGREAT
LESSA:
        ..
        ..
        ..
        ..
```

The first instruction 'ComPare the B register with the A register' subtracts the B register from the A register. The actual result is not updated to the accumulator and only affects the flags.

If the A register was equal to the B register then the Zero flag would be set, causing the program to jump to the address labelled EQUAL. If the they were not equal, then the program would carry on to the next instruction:

```
JR C,BGREAT
```

If the Carry flag was set this would indicate that the B register was greater than the Accumulator, causing a branch to the label BGREAT. If no branch occurred this would mean that the B register was less than the A register, causing the program to arrive at the label LESSA.

```
CP n          CP (HL)
CP A          CP (IX+dd)
CP B          CP (IY+dd)
CP C
CP D
CP E
CP H
CP L
```

Set Bit and Reset

There are other bit instructions in the Z80 set which allow us to set, reset or test individual bits in a byte.

*The **SET** instruction*

The 'SET' instruction allows us to SET a particular bit in a byte. We can test individual bits in a register or a RAM location. The format of the SET instruction can be any of the following forms:

```
SET n,r
SET n,(HL)
SET n,(IX+dd)
SET n,(IY+dd)
```

where n is the bit number we wish to test 0–7 and dd is an offset in the range −127 to 128

For example, the instruction:

```
SET 4,A
```

would set bit 4 of the Accumulator.

We can also use the indexing addressing mode to set and reset bits. If, for example, the IY register pointed to the address 6000H and the contents of its adjacent memory locations were as below:

Address	Contents
6000H	22H
6002H	00H
6002H	08H

Then the instruction SET 4, (IY+2) would have the following effect:

address	contents
6000H	22H
6001H	00H
6002H	18H

The contents of location 6002 hex are changed to 18 hex=24 decimal

*The **RES** instruction*
This has the opposite effect to the SET instruction; it RESets a bit in a byte or RAM location.

*The **BIT** instruction*
The bit instruction allows us to test for individual bits of a register or byte. The results of the test are signified by resetting or setting the Zero flag. If the bit tested was zero then the Zero flag would be set and, if not, the Zero flag would be reset.

```
BIT 7,A
```

The above instruction would read 'test BIT 7 of the A register'. Therefore, if the A register contained 128, which is 10000000 binary,

then the instruction would reset the Zero flag as bit seven is set to 1. If, however, we used the instruction:

```
BIT 0,A
```

with the same contents in the accumulator the Zero flag would be set, as bit 0 is zero.

The BIT instruction is very useful because it does not corrupt anything we are testing. Similar to the COMPARE instructions, it affects the bits in the flag registers only.

Spectrum INs and OUTs

The Z80 chip needs to interface to other devices such as the keyboard and a cassette recorder so that the user can communicate with the computer. There are two methods what we can use to communicate to these devices. One is known as memory mapping, that is PEEKing or POKEing, the other is by PORT addressing. A PORT is a gateway to these devices which can be read by using the instruction 'IN' or written to by using 'OUT'. There are 256 of these PORTS on the Sinclair computer. Most can be used by electronics buffs, to link up to devices such as speech synthesisers and sound chips.

There are two instructions in BASIC, 'IN' and 'OUT' which allow us to gain access to these ports. Frequently, these instructions are used to scan the keyboard or output to the speaker to produce noises.

The keyboard is divided into 8 rows of 5 keys each and the actual syntax of Spectrum BASIC to read the keyboard uses a two byte number. For example:

```
LET X=IN 61438
```

scans the keys 0 to 6 on the top row of the keyboard. The other addresses and the keys they scan are given below:

ADDRESS	HEX	KEYS SCANNED
32766	7FFE	SPACE,SYMBOL SHIFT,M,N,B
49150	BFFE	ENTER,L,K,J,H
57342	DFFE	P,O,I,U,Y
61438	EFFE	0,9,8,7,6
63486	F7FE	1,2,3,4,5
64510	FBFE	Q,W,E,R,T
65022	FDFE	A,S,D,F,G
65278	FEFE	CAPS SHIFT,Z,X,C,V

So if we wanted to scan for the bottom row of keys from SPACE to the letter B we would use the BASIC instruction:

```
LET X=IN 32766
```

A value is returned in the variable X depending on which keys are pressed. There are five bits which represent the state of each row on the keyboard. If a particular bit is low (i.e. Ø) then this means that a key is depressed, and if no keys were depressed then all bits would be high. In the table above the key values have been given in bit order, so if we were scanning the keys 1 to 5 then bit Ø would indicate the state of the key 1 ,bit 1 the state of key 2 and so on.

The other 3 bits returned when scanning the keyboard are not used and are unpredictable (mainly because of the different models of spectrums available) so it is wise not to compare the values read unless you mask out the first five bits. Masking means removing bits according to a pattern. To mask the top three bits you would have to do RES 7, r RES 6, r and RES 5, r to set them all to zero.

*The **IN** instruction*

In machine code to read a PORT we use the instruction:

```
IN A,(port)
```

The value port is the PORT address which is a one byte number in the range Ø–255. This port is read and the value is returned in the Accumulator. How do we use this instruction to scan the various lines on the keyboard? Well, if you look closely at the address which you scan in BASIC to read a particular row you will notice that the low bytes of each address are all FE hex,254 decimal. The port address and the high bytes all differ from each other.

To read a set of keys in machine code we first LOAD the Accumulator with the high byte of the line we wish to read and then execute the instruction:

```
IN A,(ØFEH)
```

So for example, if we wanted to scan the keys Ø to 6, we would write the following code:

```
LD A,ØEFH    ;SELECT LINE Ø–6
IN A,(ØFEH)  ;READ PORT
```

Now, if we wanted to test if the key Ø was pressed we could use the BIT instruction:

```
BIT 0,A  ;Test for "0"
```

This would set the Zero flag if the key was pressed or reset the flag if it was not pressed.

It is an easy matter to read a set of keys by using the compare instruction:

```
LD  A,0FBH    ;select keys Q to T
IN  A,(0FEH)  ;read key board port
AND 31        ;mask off lower 5 bits
```

This portion of code sets the Zero flag if all the keys Q,W and E are pressed.

There is another form of the IN instruction which allows us to specify the port by the value in the C register. The register in which the value is read can also be chosen from the set A,B,C,D,E,H or L.

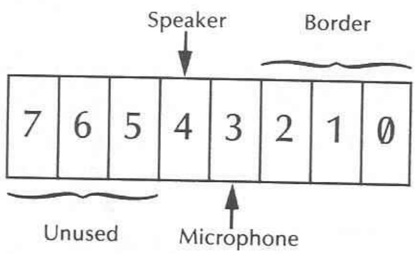
The code:

```
LD  C,0FEH
LD  A,0FBH
IN  E,(C)
```

reads the line Q to T and places the value read in the E register.

*The **OUT** instruction*

The 'OUT' instruction is often used to generate sound and output to the cassette system. The port which controls the small pisa speaker is at the address 0FEh,254 decimal. This also has the ability to change the screen border colour. The values of output to this port is in the format shown below:



PORT 254

The first three bits (bits 0–2) of the byte are used for the border colour. Bit 3 is used to control the EAR and MIC sockets so that data

can be sent and read to and from a cassette unit. Bit 4 is used to pulse the (so-called!) speaker in the Spectrum.

Sound generation on the Spectrum is a simple matter of pulsing bit 4 of port 254 high and then low for a short period of time. We would set bit 4 (i.e. to 1) and hold it high for a short time and then set it low (Ø) and hold it at this level for the same period of time. The delay we have between 'flipping' bit 4 of the port determines the frequency or note we get from the speaker. A long delay produces a low frequency and a short delay a high frequency. To output the value in the Accumulator to a port we use the mnemonic:

```
OUT (addr),A
```

This simply reads 'OUTput the value in the Accumulator to the port address' So to turn the speaker on we would use the instructions:

```
LD     A,16
OUT    (ØFEH),A
```

Notice how we first LOAD the Accumulator with 16. All this does is to set bit 4 high, which when sent to the port turns the speaker on. The following program demonstrates how the OUT instruction can work to generate sound. Both the assembler mnemonic listing and a BASIC listing have been given, with which the machine code can be loaded. Line 2Ø of the BASIC program changes the low bytes of the values for the duration and the frequency.

Assembler Listing

```
        ORG 28000D
        JP NOISE                 ;JUMP TO MAKE A NOISE
SOUND:
        LD A,10H                 ;MASK SPEAKER
                                 ;SO BIT 4 IS HIGH
        OUT (0FEH),A             ;TURN ON SPEAKER
        CALL  DELAY              ;AND KEEP HIGH
                                 ;FOR A SHORT WHILE
        XOR A                    ;TURN BIT 4 OFF
        OUT (0FEH),A             ;TURN SPEAKER OFF
        CALL DELAY               ;AND KEEP IT OFF
                                 ;FOR A SHORT WHILE
        RET
```

```
DELAY:
        LD B,D                  ;TRANFER DE TO
                                ;BC REGISTER PAIR
        LD C,E                  ;IE.PLACE DELAY IN
                                ;BC REGISTER

LOOP:
        DEC BC                  ;DECREMENT BC REGISTER PAIR
        LD A,B                  ;AND REPEAT
        OR C
        JR NZ,LOOP              ;UNTIL BC PAIR IS ZERO
        RET                     ;RETURN AFTER FINISHING
                                ;DELAY


DELA   EQU 100                  ;DELAY
DURAT  EQU 100                  ;DURATION

NOISE:
        LD DE,DELA              ;GET DELAY
        LD HL,DURAT             ;GET DURATION
BUZZ:   CALL SOUND              ;MAKE A SOUND USING
                                ;DELAY AND DURATION
        DEC DE                  ;SUBTRACT ONE OFF DELAY
        DEC HL                  ;SUBTRACT ONE OFF DURATION
        LD A,L
        OR H
        JR NZ,BUZZ              ;REPEAT SOUND UNTIL
                                ;DURATION IS ZERO.
        RET


        END
```

BASIC Program Listing

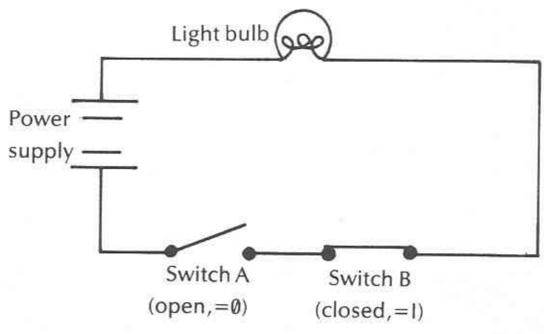
```
10 FOR A=1 TO 100
20 POKE 28029,A: POKE 28026,A
30 FOR X=1 TO 20
40 RANDOMIZE USR 28000
50 NEXT X
60 NEXT A
```

5 Operating logically

There are three logical operations available in the Z80 instruction set: AND, OR and XOR. These are all 8 bit operations which are best explained by looking at a series of diagrams and what are known as 'Truth Tables'. As explained in Chapter One numbers are represented in the computer by a series of 0's and 1's called bits and the Z80 groups 8 of these bits together to form a byte. Logical operations are performed on all 8 bits of a byte, and transform their values as described below.

AND Operations

The AND function operates on the corresponding bits of two bytes. If both corresponding bits were 1, then our result (another byte) after 'ANDing' these two bytes would set that bit to 1. If either or both of the bits were zero then the resulting bit would be zero. Look at the simple circuit diagram below:



This diagram has two switches, labeled A and B, a power supply and a light bulb. If we take the state of a closed switch as representing 1 and the open state as 0, then in order to switch the light on we have to have both switches on, i.e. then are both set to 1. We can represent the possible combination of the switches and their effective result on the light bulb by a truth table. The final result is 1 if the light bulb lights and 0 if it doesn't.

AND TRUTH TABLE

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

As you can see Switch A and switch B have to be closed to make the light bulb light up.

When we do a logical operation in machine code all the operations act on the Accumulator. The AND operator is useful for picking up bits which we want to examine. This is known as 'masking'. If we had the code:

```
LD A,01001010B
AND 00000111B
```

Since the data is in binary we require the 'B' suffix after the numeric value. The second operation is the AND function. It takes each bit of the data and AND's it with the corresponding bit in the accumulator:

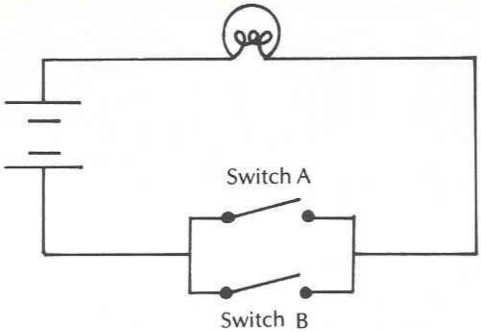
```
        01001010
AND     00000111
        --------
        00000010=2 decimal
```

AND instruction set

```
AND n    AND (HL)
AND A    AND (IX+dd)
AND B    AND (IY+dd)
AND C
AND D
AND E
AND H
AND L
```

OR Operations

The OR function is analagous to the circuit diagram below:



As you can see either one of the switches can be on to set the light on. Now looking at the possible combinations of switching in this circuit we get the following truth table:

OR TRUTH TABLE

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

The OR operator is useful to set a series of bits in the Accumulator.

```
LD A,10101101B
OR 11100000B
```

The above code would set the top three bits in the Accumulator.

```
        10101101
OR      11100000
        --------
        11101101=237 decimal
```

***OR** instruction set*

```
OR n      OR (HL)
OR A      OR (IX+dd)
OR B      OR (IY+dd)
OR C
OR D
OR E
OR H
OR L
```

XOR Operations

The XOR 'exclusive OR' operator is a little more difficult to explain with the aid of a circuit diagram but is just as easy to understand. It is similar to the OR operator but either, not both bits, may be high to give a high output

XOR TRUTH TABLE

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

The XOR operator is used to complement bits in the Accumulator and is sometimes known as *toggling*. What was previously on would be turned off, and what was off would be turned on. This instruction would be ideal for turning lights on and off connected to a computer. If, for example, we had the location at the address labelled LIGHT linked to some hardware which turned on a light if it contained 1 and turned the light off if it contained a zero, then the following program would generate a flashing strobe:

```
        LD B,0          ;set delay
        LD A,1          ;set state of switch
TOG:    XOR 1           ;toggle switch
        LD (LIGHT),A    ;turn light on or off
DELAY:  DJNZ DELAY      ;short delay
        JR TOG
```

The B register is loaded with zero which is used as a counter in a delay loop to hold the light on or off for a short period of time.

Because the DJNZ instruction will decrement B before testing it, giving B a start value of 0 causes 256 loops round the DJNZ.

```
                LD A,1
```

The Accumulator is then set to 1 and toggled, leaving zero in the Accumulator since 1 XOR 1=0

```
TOG:            XOR 1
                LD (LIGHT),A
```

The LOAD instruction turns the light on or off according to the result in A, so first time round this would turn the light off. We now leave the light in this state for a period of time using a DJNZ instruction which counts from zero to 255 then back to zero again:

```
DELAY:          DJINZ DELAY
```

We now come to the last instruction which transfers program control back to the address TOG:

```
                JR TOG
```

This time, with A register containing zero the exclusive OR function will set the A register to one, producing a strobing effect.

***XOR** instruction set*

XOR n	*XOR (HL)*
XOR A	*XOR (IX+dd)*
XOR B	*XOR (IY+dd)*
XOR C	
XOR D	
XOR E	
XOR H	
XOR L	

6 Block manipulation

Block instructions give the Z8Ø the ability to move or compare blocks of data automatically or semi-automatically. A feature not found on any other 8 bit microprocessor on the market today.

A common use of block move instructions is to reduce screen flicker in games. A technique I will show you later. First, let us look at the block compare instructions. There are four instruction concerned with searching for a particular value ('key') in a block of data. To search for this key we can use any one of the following instructions:

CPIR
CPDR
CPI
CPD

CPIR

The A register is LOADed with the value which we are searching for (the 'key'). The HL register pair is LOADed with the address of the start of the block we wish to search, and the BC register pair is set up to contain the number of bytes we want to search through. The CPIR instruction is used to automatically go through all the data, comparing the contents at each address until it either finds the key it is searching for or until it has exhausted the search. This is signified by the BC register containing Ø.

If the key is found then the Zero flag is set and the HL register pair points to the next address after the key. So the CPIR can be thought of as three instructions INC HL, CP (HL) and DEC BC.

Take a look at the following example:

```
STRING:        DEFM  "ABCDEFGH"
SEARCHFORWD: LD    HL,STRING
               LD    BC,8
               LD    A,"G"
               CPIR
               JR    Z,FOUND
NOTFOUND:      .
               .
               .
FOUND:         DEC   HL
```

The first line contains the assembler psuedo operator DEFM 'DEFine Message' which tells the assembler to place the string "ABCDEFGH" in memory when the program is being assembled. As you can see the HL is LOADed with the start of the string and BC the number of bytes we wish to search through. The Accumulator contains the key "G", which we want to seek. The CPIR instruction will find this key (as it is contained within the string) and cause the program to jump to the address at the label FOUND. At this point we subtract the HL register by one to point to the actual address where the key is.

CPDR

The CPDR instruction is similar to the CPIR instruction but the HL register pair points to the end of the block of data and the search is made backwards. This time when a key is found the Zero flag is set as before, but the HL register pair will point to one less than the address where the key was found.

```
STRING:        DEFM  "ABCDEFGH"
SEARCHBACK:    LD    HL,STRING+8
               LD    BC,8
               LD    A,"G"
               CPIR
               JR    Z,FOUND
NOTFOUND:      .
               .
               .
FOUND:         INC   HL
```

As you can see, our code for searching for a key is similar to the last subroutine SEARCHFORWD. However we start off with LOADing the HL register pair with the address of the end of the string:

```
LD HL,STRING+8
```

This kind of instruction is allowed on most assemblers and all it does is to add the offset (+8) to the address of the label to get the resultant address. When the key is found it arrives at the label FOUND. However, this time the HL register pair will point to the character "F" so to correct this we amend the HL register pair with instruction:

```
FOUND:          INC     HL
```

CPI and CPD

These are known as semi-automatic instructions. If we use a CPI instruction then it will compare the A register with the contents of the HL register. The HL register pair will be incremented and the contents of the BC pair will be decremented. Flags will be set according to the result of the comparison and the subtraction of the BC pair. The two most significant flags to test are the PO flag (Parity Odd) and the Zero flag. If the key is found then the Zero flag is set. On the other hand if the search is exhausted and the key is not found then the P/V flag is set. These instructions are useful when we are searching through non-continuous data. For example, if we wanted to search though a string and the data we are seeking occurs every three bytes of the string, then we could use the following code:

```
        LD   HL,STRING      ;SET OF START OF STRING
        LD   BC,LENGTH      ;SET LENGTH
        LD   A,KEY          ;KEY TO SEARCH FOR
LOOK:   CPI                 ;COMPARE (HL) WITH A REG
                            ;DECREMENT BC
                            ;AND INCREMENT HL
        JP   Z,FOUND        ;FOUND KEY
        JP   PO,NOTFOUND    ;EXHAUSTED SEARCH,NOT FOUND
        INC  HL             ;SKIP PASS
        INC  HL             ;UNWANTED DATA
        JR   LOOK           ;KEEP SEARCHING
```

The routine will exit to the memory address specified by the label FOUND if the key is in the string or to that given by label NOTFOUND if the key is not in the string. Notice that there are only two INC HL instructions, not three. This is because the CPI instruction has already incremented the HL pair.

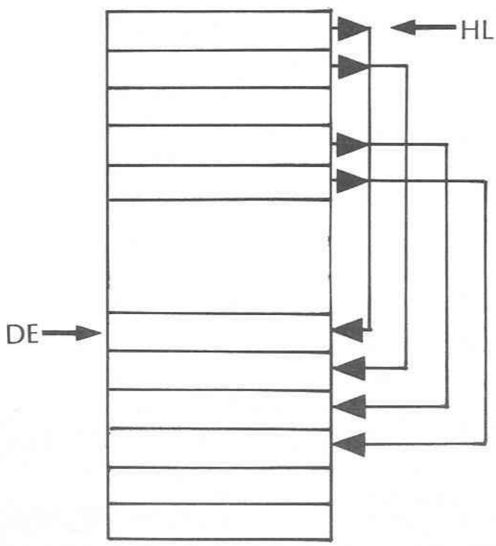
Block transfer

The Z80 is unique amongst 8 bit microprocessor chips in possessing a set of block transfer instructions. These allow blocks of data to be

moved around within memory utilising just four instructions. The HL register is LOADed with a 16 bit address which points to the start of the block to be moved. The BC register contains the numbers of bytes in that block which we wish to move. The DE register contains the destination address where the first byte of data is to be stored. After setting up these registers we could use any one of four block move instructions. Like the Block Compare set of instructions the Block Transfer possibilities allow for two automatic and two semi-automatic instructions. The two instructions LDIR (LOaD Increment and Repeat) and LDDR (LOaD Decrement and Repeat) are the two automatic block instructions. They allow us to move whole blocks of memory simply by executing the instruction once.

LDIR

The LDIR instruction is used to move data which is held in a continuous sequence of memory locations. The HL registers are set up to point to the start of the data block, the DE pair is set up to point to the start of the destination and the BC register the number of bytes to move. When executing the LDIR instruction the contents of the location pointed by the HL pair is copied to the location pointed to by the DE pair. Both the HL and the DE register pairs are incremented to point to the new data and destination locations while the BC register pair is decremented. This transfer continues until the BC register pair reaches zero, then the Z80 goes onto the next instruction.

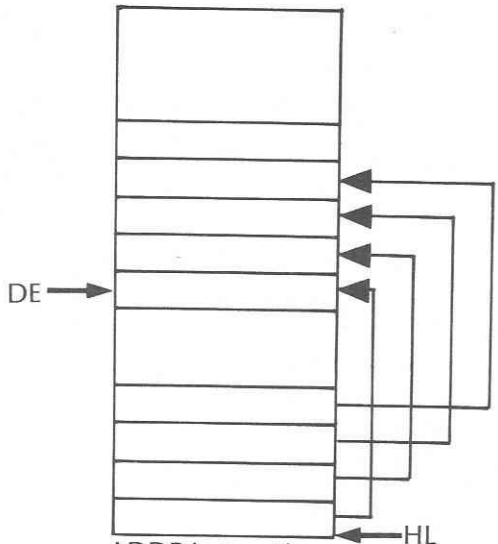


Block move instruction LDIR instruction

LDDR

The LDDR instruction is similar to the LDIR but the HL register pair and

DE register pair point to the end of the block and destination addresses. The transfers are made as with the LDIR instruction but the HL and DE pairs are decremented to point to the new data and destination addresses. Again the BC is decremented and the transfer continues until the BC register has reached zero.



LDDR instruction

Most games programmers use the block move instructions to move vast amounts of data to the screen. When a lot of information is needed to be drawn to the screen this can result in the TV display flickering. To reduce this flicker it is possible to draw the data on a dummy screen unseen by the player. This dummy screen can then be moved to the actual screen using the block move instruction.

If you imagine that we have set up a screen full of data at the address C000 hex and we wish to move it to the screen which is at the address 16384 or 4000 hex. We might use the following code:

```
;set up dummy screen

LD      HL,C000H ;HL points to dummy screen
LD      DE,4000H ;DE points to real screen
LD      BC,1B00H ;BC contains number of bytes
LDIR             ;move it!
```

The HL register points to the start of the dummy screen which is C000 hex and the DE is set up to point to the start of screen. The BC register is set up to contain 1B00 hex or 6912 decimal, the number of bytes

contained in the display file. When we execute the LDIR instruction it moves 6912 bytes starting from the location C000 hex to the screen, reducing screen flicker to a minimum.

The two semi-automatic instructions LDI (LoaD and Increment) and LDD (LoaD and Decrement) are used similarly to the CPI and CPD instructions when the data is non-continuous or we wish to stop moving data on certain conditions.

The parity odd flag is affected by the two instructions indicating that BC has reached zero when the PARITY ODD flag is set. If we wanted to write a routine which would move a block of data to the screen from C000 hex. until we reach a zero byte, then the following code could be used:

```
        LD    HL,C000H ;point to start address
        LD    DE,4000H ;point to destination address
        LD    BC,1B00H ;maximum number of bytes to move
MOVE:   LDI            ;move one byte (HL)——>(DE)
                       ;DE=DE+1:HL=HL+1:BC=BC-1
        RET   PO       ;PARITY ODD FLAG set, all done
        LD    A,(HL)   ;get next contents
        AND   A        ;test for zero
        JR    NZ,MOVE
        RET            ;return we have reached a zero!
```

The routine will move at least one byte as the test for a zero byte is made after the LDI instruction. The transfer is complete if the PARITY ODD flag is set, indicating that we did not encounter a zero byte in the dummy screen and the BC register reached zero. It will exit when we reach the first zero byte. The AND A is used here to test for zero. A more obvious method would be CP0 (ComPare with zero) but AND A is more efficient as it is faster to execute and uses less memory space. The AND A will leave the contents of A unchanged since any bit that is AND'ed with itself will remain unchanged (see the Truth Table for AND in chapter 5). The AND instruction will set the flags according to the eventual contents of A so this is an easy way of setting flags. OR A would be equally suitable for this purpose but XOR A would clear the A register to zero.

Miscellaneous instructions

The next (and last) batch of five instructions that will be explained in this book all operate on the Accumulator or flag register, so they have been grouped

***CPL** complement accumulator*
The Complement instruction simply replaces 0's for 1's and vice versa.
For example:

```
LD A,187 (10111011 binary)
CPL
```

The A register will contain 68 (01000100 binary) after executing the Complement instruction.

***NEG** Negate accumulator*
The Negate instruction has the effect of multiplying the number by −1. It changes the number's sign (not just the sign bit!)
For example:

24 hex (36 decimal) becomes DC hex (−36 decimal)

This instruction performs the 'two's complement' on the contents of the Accumulator. It is directly equivalent to the pair of instructions:

```
CPL
INC A
```

***CCF** Complement Carry Flag*
This changes the Carry flag to a 1 if it was a 0 and vice versa.

***SCF** Set Carry Flag*
This instruction forces the Carry flag to a 1.

***DAA** Decimal Adjust Accumulator*
This instruction is used to add numbers which are represented in Binary Coded Decimal (BCD) form. The decimal numbers 0 to 99 can be represented in one byte by splitting it into two sets of 4 bits each, called nybbles. The left nybble is the number of tens in the number and the right nybble represents the number of units. For example, the number 29 can be represented by the BCD number 0010 1001 (the 8 bit binary number has been split into two nybbles to make it easier to read.)

When we want to add or subtract two BCD numbers we use the normal ADD or SUB instructions followed by the DAA instruction. The H and N flags are used by the DAA to adjust the result to BCD.
For example:

```
LD    A,29H    ;LOAD A with 29 hex 41 decimal 29 BCD
LD    B,24H    ;LOAD B with 24 hex 36 decimal 24 BCD
ADD   A,B      ;ADD B register to A register
DAA            ;decimal adjust
```

This piece of code would leave the result 53 hex in the A register (not 4D hex as with normal addition).

The way that DAA works is that after an arithmetic operation it checks whether the low nybble is in the range 0 to 9. If this is not the case, it will add 6 to the low nybble, which causes the high nybble to be incremented. Then the high nybble is checked. If it exceeds 9 then 6 is added to the high nybble, which will overflow into the Carry flag.

In the example the Accumulator will hold the value 4DH before DAA is executed.

Here is the flow of logic for DAA in this case:

(a) Low nybble=DH
(b) This is greater than 9 so add 6 to the low nybble to give temp. result of 13H.
(c) replace the low nybble of temp. result to leave 10H or the high nybble equal to 1.
(d) Add the high nybble of the temp. result to the high nybble of the accumulator to give:

$$4+1=5$$

(e) 5 is less than 9 so replace high nybble in accumulator.
(f) The Accumulator now contains 53H,this is the correct BCD result of 29+24

7 A Spectrum monitor

This chapter presents a program which will allow you to write, run and debug machine code programs. It can be entered into your Spectrum using the BASIC machine code loader given as Listing 1 below. After keying in and SAVEing the BASIC program on tape, begin the program by entering the start address at which the machine code program will start to be built up. Then INPUT the hexadecimal data which makes up the program as given in the hexadecimal listing (Listing 2). If at any time you wish to correct a mistake there are edit facilities to help you (see below). Typing $$ when prompted for a hexadecimal byte allows you to change the address at which the next piece of data is to be placed. To quit the program, type in a double hash ## when prompted for hexadecimal data. After this has been done the program allows you to store the machine code using the SAVE command.

For anyone interested in the way in which the monitor was written the corresponding assembly mnemonic listing has been included in Appendix 3. At the moment don't worry about understanding how it works, just type in the data. Once it is up and working you can use it to enter the other machine code programs which are given. In this book each functional program, other than illustrative examples, has two listings. One is in mnemonic form which is easier to read and follow. This can be used by those of you that have full assembler programs available. The second listing is the hexadecimal equivalent. This is the portion of memory of your Spectrum which holds the program. It is displayed as a hexadecimal dump and can be reproduced by keying in the appropriate value for each memory location using the hex monitor. You could, of course, use the BASIC monitor to input the machine code listings. However, as you will see this is not as powerful as its machine code counterpart.

The monitor program has been assembled at the address 25500. This allows us to write machine code programs higher up in memory, giving more free space in which to RUN both BASIC and machine programs. The monitor offers the machine code programmer eleven functions. These are: Dump, Edit, Fill, Goto, Hunt, Identify, Load, Move, Print, Register and Save.

More commands can easily be added by changing a command table to point to the routine which deals with the new command.

After the monitor has been typed in using the BASIC loader the machine code can be SAVEd by typing:

SAVE "SMON" CODE 25500, 1500.

The Spectrum is then cleared by switching the machine off and on again. Next the monitor can be loaded by typing in:

CLEAR 24999:LOAD "" CODE:RANDOMIZE USR 25500.

The monitor should then welcome you with a '>' prompt, inviting the input of one of the eleven commands.

***D** address (Hexadecimal dump)*

Type in 'D' followed by a two byte hexadecimal number. The monitor then displays the contents of memory from the given address in a hexadecimal format. The routine will keep dumping the memory contents until a key is input other than a carriage return. For example:

```
>D 0100
```

The above command will display 64 bytes of memory from the address 0100 hex.

***E** address (Edit)*

This allows you to *edit* or modify a byte in memory. To execute the command, type in 'E' followed by a two byte address which you wish to start modifying. The monitor will then show the address which is being modified and the contents of that address. Then type in a one byte hexadecimal number to change that location. After the modification has been given the monitor will automatically go onto the next location to be edited. The routine can be exited by typing in a non-valid hexadecimal digit. eg:

```
>E C000
C000 FF 3E
C001 00 2A
C002 00 C9
C003 00 <ENTER>
```

Pressing <ENTER> will exit from the edit command.

F start address end addres byte (FILL memory with byte value)
To FILL a block of memory with a given byte, given the start address, end address and byte value to where the byte is to be filled. For example:

```
>F 4000 5800 2A
```

This will fill the memory from 4000 hex to 5800 hex with the byte 2A hex.

G address or G address, breakpoint address (GOTO address)
This command allows execution of a portion of code from a given address. A second parameter can be given which allows you to give a break point where the register values will be displayed.

To give a breakpoint type in a ',' after the first address and then type in the breakpoint address.

```
>G C000
```

The above example will cause the monitor to execute from address C000 and will RETURN back to the monitor after a RET instruction is met.

```
>G C000 ,C004
```

This second example will cause the monitor to execute from the address C000 with a breakpoint at C004. If the code flows through this address then it will return to the monitor displaying the Breakpoint address and the contents of the registers. Such an example could be as follows:

```
*C004

AF    BC    DE    HL    IX
2A3E  22AA  0000  DEF0  DDFE
0000  0000  2232  2312
```

H start address end address byte value (HUNT for a byte)
The HUNT command will allow you to search for a specific byte through a given set of addresses. Type in 'H' and then give the start address, the end address and the byte value for which you wish to search. The routine will then display each address where that byte is found, pausing for you to type in ENTER. To exit from this routine before the search is exhausted, any other key may be pressed.

```
>H 0000 0100 2A
0008
0018
0031
003A
0068
007A
>
```

The above example will search for the byte 2A hex. From the address 0000 to 0100 hex. When the search is exhausted the monitor returns with the prompt.

I string (Identify file name)
This command is used in conjunction with the SAVE and LOAD command identifying the file name to be LOADed or SAVEd.

Type in 'I' and then input a filename consisting of no more then ten letters of the alphabet. Lower case letters will be ignored.

```
>I SPECMON
```

The example above will set the identifier to the string 'SPECMON'.

L start address, number of bytes (Load file)
The LOAD command will wait for the filename given by the identifier (see the 'I' command) and once found on the tape will start to LOAD it at the given address. The second parameter (also a two byte number) specifies the number of bytes to LOAD. The LOAD command will read in each file of the tape and display its header to the screen.

```
>I SPECMON
>L C000 0100
Waiting for SPECMON
```

The commands above will LOAD the file 'SPECMON' to the address C000 hex. The number of bytes to be LOADed is 256 (i.e. 0100 hex). To exit from the LOAD command at any time press both the <CAPS SHIFT> and the <BREAK> keys.

M start address, end address, destination address (MOVE Block)
This command will MOVE blocks of data to a given address. You need to specify the start address, end address and the destination address.

```
>M C000 DB00 4000
```

The above example will MOVE data from the address C000 up to DB00 hex to the address 4000 hex.

P address (PRINT ASCII)
The PRINT command is similar to the DUMP command except that it displays the contents of the memory in ASCII code form rather than hexadecimal. For example,

```
>P 0690
```

will PRINT the ASCII contents from the address 0690 hex. To continue the listing press <ENTER>, otherwise press any other key.

***R** or **R r** or **R 'r** (REGISTER modify)*
The REGISTER command allows you to examine or modify any register r, where r can be any of the following:

A	modifies the AF register pair
B	modifies the BC register pair
D	modifies the DE register pair
H	modifies the HL register pair
X	modifies the IX register pair

To examine the contents of the registers type in the command 'R' and the <ENTER>.

```
>R ENTER

AF     BC     DE     HL     IX
FF3E   0000   0000   F22A   0000
0000   0000   0000   002A
```

The second value under each register pair is the contents of the alternative register set. The IY register is not shown as this is used by the BASIC system.

To modify a register, you can simply type in the register pair you wish to modify, after typing in the command 'R'. The monitor will then display the current value of the register pair, waiting for an INPUT of the new value. To modify the alternative set, type a ' before entering the register pair. For example:

```
>R H 002A 2FFF
>R 'D 0000 2A33
```

This will change the contents of HL register pair to the value 2FFF hex, and the alternate DE register pair to 2A33 hex.

S start address ,number of bytes (SAVE)
The SAVE command is used to SAVE machine code to a tape recorder. The start address and the number of bytes to SAVE are the parameters that you will need to specify with this command. The monitor will then prompt you to get the tape recorder ready and SAVE the portion of code on tape. The file name is set by using the 'I' (Identify) command. For example:

```
>I FRED
>S 4000 1B00
Press any key when ready
```

These will SAVE the 6912 bytes from the address 16384 to the tape. The file will be SAVEd as 'FRED'.

The listings follow below.

Listing 1: BASIC Hex Monitor

```
  10 CLEAR 25499
  20 CLS : GO SUB 160
  30 LET a=x
  40 GO SUB 320: PRINT x$;":";
  50 FOR z=1 TO 8: GO SUB 90: IF
 a$="$$" THEN GO TO 10
  60 LET a=a+1: NEXT z
  70 PRINT
  80 LET x=a: GO TO 40
  90 INPUT "hex :"; LINE a$: IF
LEN a$<>2 THEN GO TO 90
 100 IF a$="$$" THEN RETURN
 110 IF a$="##" THEN STOP
 120 GO SUB 250
 130 IF e=1 THEN GO TO 90
 140 POKE a,x: PRINT a$;" ";
 150 RETURN
 160 INPUT "addr:"; LINE b$: IF
LEN b$<>4 THEN GO TO 160
 170 GO SUB 250
 180 IF e=1 THEN GO TO 160
 190 LET x=t*256+x: LET a=x
 200 RETURN
 210 REM two byte hex input
 220 LET a$=b$(1 TO 2)
```

```
 230 GO SUB 250: LET t=x
 240 LET a$=b$(3 TO 4)
 250 REM one byte hex input
 260 LET e=0
 270 LET l=FN x(2): IF l>15 THEN
 LET e=1
 280 LET h=FN x(1): IF h>15 THEN
 LET e=1
 290 LET x=h*16+l
 300 RETURN
 310 DEF FN x(n)=CODE a$(n)-48-(
CODE a$(n)>57 AND CODE a$(n)<71)
*7-(CODE a$(n)>96 AND CODE a$(n)
<103)*39
 320 REM two byte input
 330 LET h=INT (x/256): LET l=x-
h*256
 340 LET x=h: GO SUB 370
 350 LET x=l: GO SUB 380
 360 RETURN
 370 LET x$=""
 380 LET p=INT (x/16): GO SUB 39
0:LET p=x-(INT (x/16))*16
 390 REM hex
 400 IF p>9 THEN LET a$=CHR$ (p+
CODE "A"-10)
 410 IF p<=9 THEN LET a$=CHR$ (p
+CODE "0")
 420 LET x$=x$+a$
 430 RETURN
```

Listing 2: Spectrum Monitor Hexadecimal Listing

```
639C   C3  36  64  3E  FF  CD  C2  04
63A4   C9  37  3E  FF  CD  56  05  C9
63AC   11  11  00  DD  21  06  69  AF
63B4   37  CD  56  05  3A  11  69  4F
63BC   3E  24  32  11  69  CD  CD  68
63C4   11  07  69  CD  2C  64  3E  20
63CC   CD  22  64  EB  71  23  23  23
63D4   7E  CD  92  66  2B  7E  CD  92
```

63DC	66	2B	3E	20	CD	22	64	7E
63E4	CD	92	66	2B	7E	CD	92	66
63EC	CD	CD	68	C9	11	11	00	DD
63F4	21	06	69	AF	CD	C2	04	C9
63FC	E5	C5	D5	CD	A7	64	3A	3B
6404	5C	CB	6F	28	F9	CB	AF	32
640C	3B	5C	D1	C1	E1	C9	CD	FC
6414	63	3A	08	5C	CD	22	64	C9
641C	3E	02	CD	01	16	C9	F5	F5
6424	AF	32	8C	5C	F1	D7	F1	C9
642C	1A	FE	24	C8	CD	22	64	13
6434	18	F6	31	6E	69	CD	1C	64
643C	11	B0	65	CD	2C	64	11	44
6444	65	CD	2C	64	CD	CD	68	31
644C	6E	69	3E	08	32	6A	5C	21
6454	6F	69	36	4B	23	36	64	2B
645C	22	3D	5C	21	5F	64	E5	CD
6464	CD	68	3E	3E	CD	22	64	CD
646C	12	64	D6	41	D8	FE	13	D0
6474	87	21	81	64	5F	16	00	19
647C	5E	23	56	EB	E9	83	66	83
6484	66	83	66	1A	66	B3	64	E1
648C	66	17	67	8F	68	5E	68	83
6494	66	83	66	FC	64	33	68	83
649C	66	83	66	5A	66	83	66	E2
64A4	67	B5	65	01	00	80	11	00
64AC	40	21	00	40	ED	B0	C9	CD
64B4	EE	64	CD	8D	66	3E	20	CD
64BC	22	64	7E	CD	92	66	3E	20
64C4	CD	22	64	E5	CD	B1	66	E1
64CC	77	23	CD	CD	68	18	E0	CD
64D4	D6	66	E5	3E	20	CD	22	64
64DC	CD	D6	66	E5	D1	E1	C9	CD
64E4	D3	64	E5	D5	CD	EE	64	D1
64EC	E1	C9	3E	20	CD	22	64	CD
64F4	D6	66	E5	C1	CD	CD	68	C9
64FC	3E	20	CD	22	64	CD	D3	64
6504	22	40	65	7B	B2	CA	AE	66
650C	ED	53	42	65	11	7D	65	CD
6514	2C	64	11	17	69	CD	2C	64

651C	CD	AC	63	11	07	69	21	17
6524	69	06	0A	1A	CB	AF	4E	CB
652C	A9	B9	20	EC	23	13	10	F3
6534	ED	5B	42	65	DD	2A	40	65
653C	CD	A5	63	C9	00	00	00	00
6544	0D	2A	53	42	55	47	2A	20
654C	28	43	29	20	4A	6F	68	6E
6554	20	57	69	6C	73	6F	6E	20
655C	31	39	38	34	2E	0D	24	0D
6564	50	72	65	73	73	20	61	6E
656C	79	20	6B	65	79	20	77	68
6574	65	6E	20	72	65	61	64	79
657C	24	0D	57	61	69	74	69	6E
6584	67	20	66	6F	72	20	24	0D
658C	52	4F	55	54	49	4E	45	20
6594	4E	4F	54	20	49	4D	50	4C
659C	45	4D	45	4E	54	45	44	24
65A4	0D	2A	2A	45	52	52	4F	52
65AC	2A	2A	0D	24	16	01	01	0D
65B4	24	3E	20	CD	22	64	CD	D3
65BC	64	22	13	69	7B	B2	CA	AE
65C4	66	ED	53	11	69	11	63	65
65CC	CD	2C	64	CD	A7	64	CD	A7
65D4	64	CD	A7	64	CD	12	64	3E
65DC	03	11	06	69	21	17	69	12
65E4	13	01	0A	00	ED	B0	CD	F0
65EC	63	CD	A7	64	CD	A7	64	CD
65F4	A7	64	DD	2A	13	69	ED	5B
65FC	11	69	CD	9F	63	C9	E5	CD
6604	92	66	3E	20	CD	22	64	E1
660C	C9	06	08	7E	CD	02	66	23
6614	10	F9	CD	CD	68	C9	3E	20
661C	CD	22	64	CD	EE	64	0E	08
6624	CD	8D	66	3E	20	CD	22	64
662C	CD	22	64	CD	0D	66	0D	20
6634	EF	CD	CD	68	CD	CD	68	CD
663C	12	64	FE	0D	28	E0	C9	06
6644	15	7E	FE	20	38	04	FE	80
664C	38	02	3E	2E	CD	22	64	23
6654	10	EF	CD	CD	68	C9	3E	20

665C	CD	22	64	CD	EE	64	0E	08
6664	CD	8D	66	3E	20	CD	22	64
666C	CD	22	64	CD	43	66	0D	20
6674	EF	CD	CD	68	CD	CD	68	CD
667C	12	64	FE	0D	28	E0	C9	00
6684	D5	11	8B	65	CD	2C	64	D1
668C	C9	7C	CD	92	66	7D	5F	CB
6694	3F	CB	3F	CB	3F	CB	3F	CD
669C	A1	66	7B	E6	0F	C6	30	FE
66A4	3A	FA	AA	66	C6	07	CD	22
66AC	64	C9	C3	4B	64	CD	12	64
66B4	CD	C8	66	5F	CD	12	64	CD
66BC	C8	66	CB	23	CB	23	CB	23
66C4	CB	23	B3	C9	A7	DE	30	FE
66CC	0A	D8	A7	DE	07	FE	10	30
66D4	D9	C9	CD	B1	66	F5	CD	B1
66DC	66	6F	F1	67	C9	3E	20	CD
66E4	22	64	CD	D3	64	3E	20	CD
66EC	22	64	E5	EB	A7	ED	52	DA
66F4	AE	66	CA	AE	66	E5	C1	E1
66FC	E5	D1	13	E5	D5	CD	B1	66
6704	D1	E1	77	ED	B0	CD	CD	68
670C	C9	D1	21	4A	67	E5	D5	CD
6714	64	67	C9	3E	20	CD	22	64
671C	CD	D6	66	E5	3E	20	CD	22
6724	64	CD	12	64	FE	0D	28	E1
672C	FE	2C	C2	AE	66	CD	D6	66
6734	E5	11	01	69	01	03	00	ED
673C	B0	E1	36	CD	23	36	7E	23
6744	36	67	CD	64	67	C9	ED	73
674C	04	69	31	01	69	08	D9	E5
6754	D5	C5	F5	D9	08	DD	E5	E5
675C	D5	C5	F5	ED	7B	04	69	C9
6764	ED	73	04	69	31	EF	68	F1
676C	C1	D1	E1	DD	E1	08	D9	F1
6774	C1	D1	E1	08	D9	ED	7B	04
677C	69	C9	CD	4A	67	E1	2B	2B
6784	2B	CD	CD	68	3E	2A	CD	22
678C	64	CD	8D	66	EB	21	01	69
6794	01	03	00	ED	B0	CD	B4	67

679C	C9	06	04	5E	23	56	E5	EB
67A4	CD	8D	66	3E	20	CD	22	64
67AC	CD	22	64	E1	23	10	EC	C9
67B4	CD	CD	68	11	D3	68	CD	2C
67BC	64	21	EF	68	CD	9D	67	CD
67C4	CD	67	CD	CD	68	CD	9D	67
67CC	C9	7E	F5	23	7E	CD	92	66
67D4	F1	CD	92	66	23	3E	20	CD
67DC	22	64	CD	22	64	C9	3E	20
67E4	CD	22	64	CD	12	64	FE	27
67EC	20	05	CD	12	64	C6	08	21
67F4	2A	68	01	09	00	ED	B1	C2
67FC	B4	67	2B	11	2A	68	A7	ED
6804	52	11	EF	68	CB	25	19	23
680C	3E	20	CD	22	64	7E	CD	92
6814	66	2B	7E	CD	92	66	23	3E
681C	20	CD	22	64	CD	B1	66	77
6824	2B	CD	B1	66	77	C9	41	42
682C	44	48	58	49	4A	4C	50	3E
6834	20	CD	22	64	CD	E3	64	E5
683C	A7	ED	52	30	04	E1	EB	18
6844	F6	EB	C5	D5	C1	D1	F1	E5
684C	A7	ED	52	E1	38	03	ED	B0
6854	C9	09	2B	EB	09	2B	EB	ED
685C	B8	C9	3E	20	CD	22	64	CD
6864	12	64	FE	00	C8	FE	41	DA
686C	AE	66	21	17	69	06	0A	0E
6874	20	71	23	10	FC	21	17	69
687C	06	09	77	05	C8	23	CD	12
6884	64	FE	0D	C8	FE	41	DA	AE
688C	66	18	EF	3E	20	CD	22	64
6894	CD	D3	64	E5	EB	A7	ED	52
689C	DA	AE	66	CA	AE	66	E5	C1
68A4	E1	3E	20	CD	22	64	E5	D5
68AC	CD	B1	66	D1	E1	BE	F5	20
68B4	0D	CD	CD	68	CD	8D	66	CD
68BC	12	64	FE	0D	20	09	23	0B
68C4	78	B1	28	03	F1	18	E6	F1
68CC	C9	3E	0D	CD	22	64	C9	41
68D4	46	20	20	20	20	42	43	20

```
68DC    20  20  20  44  45  20  20  20
68E4    20  48  4C  20  20  20  20  49
68EC    58  0D  24  00  00  00  00  00
68F4    00  00  00  00  00  00  00  00
68FC    00  00  00  00  00  00  00  00
6904    00  00  00  00  00  00  00  00
690C    00  00  00  00  00  00  00  00
6914    00  00  00  00  00  00  00  00

691C    00  00  00  00  00  0D  24  00
6924    00  00  00  00  00  00  00  00
692C    00  00  00  00  00  00  00  00
6934    00  00  00  00  00  00  00  00
693C    00  00  00  00  00  00  00  00
6944    00  00  00  00  00  00  00  00
694C    00  00  00  00  00  00  00  00
6954    00  00  00  00  00  00  00  00
```

8 Program production

In this chapter we will go step-by-step through an example machine code program looking at the different sets of instructions and how they are assembled into machine code by the assembler. You may remember that in the first chapter we explained that there are several ways of writing machine code programs. One method is to use a professional assembler such as HI-SOFT'S 'DEVPAC'. Another is to use a monitor which allows us to INPUT machine code by its hexadecimal values.

Using an assembler is the best method of writing machine code as it is written in the mnemonic type instructions which are so easy to learn. When a program is in the process of being assembled the assembler goes through each mnemonic converting it to its machine code equivalent into another part of memory. Most assemblers on the market provide the option of seeing the mnemonic files translated to their hexadecimal equivalents and the addresses where each particular instruction is to be stored in memory.

Below is an example of an assembled listing:

```
                ORG   16384
4000    3E 21   LD    A,33
4002    C9      RET
                .END
```

The first instruction 'ORG' (ORiGin) is not a Z80 instruction but is used to tell the assembler at what address to place the first instruction in memory. Our example shows that the origin is set at 16384 decimal so that the first instruction will be assembled at 16384 or 4000 hex.

The assembled listing shows mnemonics to the right of the listing and the address and hexadecimal Op codes to the left. Why are hexadecimal values displayed and not decimals? Well why not? Hexadecimal number are used for convenience sake only. They use fewer digits to represent the decimal numbers 0–255 and are easier to read . . . well they should be easier to read after a bit of practice!

The first address is 4000 hex which is 16384 decimal. At this address the Op code for LD A, (LOAD the A register) is placed. The

content of the address 4001 hex (16385) is the data byte 33. This is shown by its hexadecimal equivalent 21 hex. Since the first instruction was two bytes long the next instruction will be placed at the address 4002 hex or 16386 decimal. This is the RETurn instruction which is only one byte long and has the value C9 hex or 201 decimal.

The last instruction 'END' is also not a Z80 instruction. This is used by most assemblers to signify that there are no more instructions to assemble. Now armed with this information we can now go through a short assembled listing looking at each instruction and the effect it has. Also listed is a hexadecimal dump of the program which can be used with the BASIC monitor or machine code monitor listed in this book. Instruction will be given on how to INPUT the machine code using the BASIC monitor. First of all let's go through the overall effect of the program.

The machine code routine enables you to enhance your programs by having a scrolling attribute BORDER along the edges of the screen. A coloured BORDER is produced along the screen by POKEing the attribute file with random PAPER colours. Then the routine begins to move the whole BORDER in a clockwise direction. The whole program comes in two parts: a BASIC listing (listing 1) and the machine code (listing 2). The BASIC program does the easy work. It draws a random coloured BORDER along the Spectrum screen. Line 10 sets the variable x to the address of the start of the attribute file 22528 and this is used to place a random line of PAPER colours along the top of the screen. Lines 20 to 40 produce the top BORDER by POKEing a random PAPER colour (The PAPER colour is produced by multiplying a random number from 0 to 7 by 8).

The lines 50 to 100 produce two coloured BORDERs along the side of the screen and finally, lines 110 to 140 produce the BORDER for the bottom of the screen. After the BORDER is produced the machine code routine is executed by the BASIC line 150 'RANDOMIZE USR 30000'. The USR command is used by BASIC to call a machine code routine in memory. The address following the USR instruction is the address to where BASIC will jump. It executes our machine routine which will shift the whole BORDER clockwise by one attribute. After the machine code routine has been executed and returns back to BASIC (by using a RET instruction) the line following the BASIC call will be executed. Line 160 is used to slow down the scroll by pausing for 1 second. Line 170 jumps back to 150 to call the machine code routine again and again until the user breaks out by pressing <CAPS SHIFT> and <BREAK SPACE> together.

Type in the BASIC program and SAVE it by typing SAVE "DEMO". DO NOT RUN it at this point, as we have not yet typed in the machine code routine. RUNning the program will probably result in the Spectrum crashing and losing the BASIC program!

With the BASIC program safely SAVEd on a cassette, you should now key-in the assembler listing on page 91. To do this you will, of course, have to LOAD your assembler first. When this listing has been entered into the Spectrum you should SAVE the source code, again onto a cassette, using the appropriate command for your assembler. Next, CLEAR the memory of the machine by switching it off and then on again.

Then LOAD up the BASIC 'DEMO'. Before re-LOADing the machine code program type in CLEAR 29000. This will re-set RAMTOP to protect our machine code program. LOAD the machine code routine from the tape into the Spectrum by typing LOAD "DEMOC" CODE and <ENTER>. After it has LOADed, RUN the program and we should get a coloured BORDER scrolling around the screen in a clockwise direction. If the program does not scroll the border then probably you have mis-typed the machine code routine. Re-type the routine again using the BASIC monitor.

Now let's look more closely at the machine code routine and see how it works:

```
ORG 30000
```

The first line tells the assembler where the origin of the machine code is to be assembled. Since our program is to be placed at the address 30000, this number is placed after the ORG instruction.

```
ATTRADD    EQU    5800H
```

The next instruction is like the ORG instruction in that it is not a Z80 one. It is used by the assembler to produce a table of strings (symbols) which hold one or two byte numbers. The string 'EQU' stands for EQUate and produces the string and gives it the value following the EQUate. In this way the above line will produce the symbol ATTRADD with the number 5800 hex (22528 decimal).

The symbol ATTRADD now holds the address of the attribute file.

```
LD  HL,ATTRADD+31   ;POINT TO RIGHT HAND SIDE OF ATTR
LD  DE,ATTRADD+30   ;DE POINTS TO THE NEXT ATTR
```

The next two instructions are at the start of the machine code program. These use the value ATTRADD contained in the assembler's symbol table. The HL register pair is LOADed with the address of 5800H

plus 31 decimal and the DE pair is LOADed with the address 5800H+ 30 decimal. The assembler will automatically calculate the two results and place the address in the source output.

The HL register pair contains the address of the far right hand corner of the top row of the attribute file. The DE register pair holds the address of the attribute location to the left of the top right hand corner (it points to the left of the HL register pair).

The object of the first portion of code is to move each of the 31 attributes of the top line of the screen one character along to the right. This is done by repeatedly replacing the attribute byte pointed to by the DE pair and placing it in the location pointed to by the HL pair. Then make the two pointers point to the next locations.

The loop DOTOP, therefore, is used to scroll the top attribute line from the left to the right:

```
        LD      A,(HL)      ;SAVE FIRST ATTRIBUTE
        PUSH    AF          ;ON THE STACK
```

Before going into this loop we have to SAVE the attribute in the top right hand corner. This will be over-written with the new attribute to its left.

The first part of the code LD A,(HL) LOADS the contents of the address pointed to by the HL pair. This is the top right hand corner of the attribute file and places the value into the A register. The second instruction PUSH AF pushes this attribute onto the stack, where it will stay until we need it. This has the effect of saving the first attribute onto the stack.

```
        LD B,31         ;LOAD B WITH COLUMN COUNTER
```

We now set up a loop counter needed to scroll the top line of the attribute file. This is done by LOADing the B register with the value 31 decimal, which is the number of characters we have to scroll across.

```
DOTOP:  LD      A,(DE)      ;GET NEXT ATTRIBUTE
        LD      (HL),A      ;AND PLACE IT TO THE ATTRIBUTE
                            ;TO THE LEFT
        DEC     DE          ;POINT TO THE NEXT ATTRIBUTE
        DEC     HL
        DJNZ    DOTOP       ;REPEAT UNTIL ALL COLUMNS DONE
```

The content of the DE register pair is LOADed into the A register with

the instruction LD A, (DE). This instruction is used to force the new byte to be scrolled to the right. This new attribute is placed to the right by LOADing it to the address pointed to by the HL pair, performed by the instruction LD (HL) ,A (LOAD into the address pointed by the HL pair the contents of the A register). The attribute, pointers are then moved to the left by one attribute by subtracting one from each of the pointers. The DECrement instructions DEC DE and DEC HL are used to implement this. The last instruction of this portion of code is the branch instruction DJNZ 'DECrement and Jump if Not Zero'. This instruction takes the B register from our column counter and subtracts one from it. If the result is not zero (if the B register does not contain a zero) then a relative jump is made to the address DOTOP. Remember, a relative jump differs from an absolute jump in that an offset number of bytes is given to where the program must jump instead of a two byte address. The assembler has the job of calculating this offset. It does this by working out the number of bytes between the lable DOTOP and the instruction DJNZ DOTOP.

After leaving this loop the program has to deal with the coloured borders along the lefthand side of the attribute file. Since the HL pair has been decremented 31 times in the loop it will now point to the start of the attribute file 5800 hex. The DE pair will point to the address 57FFH:

```
LD      BC,32     ;LET HL POINT ONE ROW DOWN
ADD     HL,BC     ;BY ADDING 32
INC     DE        ;DE   NOW   POINTS   TO   START
                  ;OF ATTRIBUTE FILE IE. 5800H
```

The HL pair is adjusted to point to the second attribute row by the adding of 32. This is done by the two instructions LD BC, 32 (LOAD the BC pair with 32) and ADD HL, BC (ADD to the HL pair the contents of the BC pair). The DE is adjusted to point to the start of the Attribute file. Since its value is 57FFH then it is a simple matter of adding one to get the desired result. Therefore we use the instruction which increments the DE pair by one: 'INC DE'.

Now we come to the portion of code which deals with scrolling the attributes from the bottom left hand side to the top left hand side:

```
LD B,21
```

The B register is set with the row counter and the loop DOLEFT is then entered. This time the HL register pair points to the new attribute and the DE pair points one row up to the old attribute. Like the first

portion of code which dealt with the scrolling, the top row of the contents of the new attribute address is placed into the old attribute address. The pointers are then updated to point to the next attributes. Since we are going down the attribute file we must add an offset of 32 to both the DE and HL register pairs. The BC register pair is placed onto the stack, which saves the row counter from being corrupted. The BC pair is then LOADed with 32 which is the offset needed to point to the next row down. This is added to the HL pair so that it now points to the next row down. It is then exchanged with the DE pair so that it too can be updated. The instruction EX DE,HL (EXchange the DE pair with the HL pair) is used because there is no such instruction as ADD DE,BC. Therefore we swap the two pairs and update the other pointer with a second ADD HL,BC instruction. To restore the registers to their new values we have to use the EXchange instruction once more. The row counter is then restored by the instruction POP BC (POP the top of the stack to the BC pair) which is decremented and tested to see if we have moved 21 bytes in the DJNZ instruction.

```
DOLEFT: LD   A,(HL)   ;GET ATTRIBUTE BELOW
        LD   (DE),A   ;AND PLACE ON OLD ONE.
        PUSH BC       ;SAVE ROW COUNTER
        LD   BC,32    ;NEXT ROW OFFSET
        ADD  HL,BC    ;HL NOW POINTS TO NEW ROW
        EX   DE,HL    ;SWOP FOR DE
        ADD  HL,BC    ;WHAT WAS DE NOW POINTS TO NEW
                      ;ROW
        EX   DE,HL    ;RESTORE BACK TO NORMAL
        POP  BC       ;RESTORE ROW COUNTER
        DJNZ DOLEFT   ;REPEAT UNTIL ALL ROWS DONE
```

After executing the above loop the DE pair points to the bottom lefthand side of the attribute file. We now need to scroll the bottom line from the right to the left. Therefore, we need the HL pair pointing to the attribute to the right of the DE pair.

```
        LD   H,D      ;PLACE DE INTO HL
        LD   L,E
```

The DE pair is first copied into the HL pair by the two instructions LOADing the high part of the DE pair (the D register) into the high part of the HL pair (the H register). This is performed with the instruction LD H,D (LOAD into the H register the contents of the D register). Then the low part is copied by using the instruction LD L,E (LOAD the L register with the contents of the E register).

Now that the HL pair is also pointing to the bottom left hand corner, point it to the right of the DE pair by incrementing it by one using instruction INC HL (INCrement the HL pair by one).

```
        INC   HL        ;HL POINTS ONE TO RIGHT OF DE
```

The next portion of code is very similar to the DOTOP but this time we are scrolling the attributes in the opposite direction. Notice that we are incrementing the pointers instead of decrementing them.

```
        LD    B,31      ;LOAD B REG WITH COLUMN COUNTER
DOBOT:  LD    A,(HL)    ;GET DATA FROM THE ATTR ON RIGHT
        LD    (DE),A    ;AND PLACE IT IN THE LEFT
        INC   HL        ;POINT TO NEW ATTRIBUTES
        INC   DE        ;TO THE RIGHT
        DJNZ  DOBOT     ;REPEAT UNTIL DONE 31 TIMES
```

After scrolling the bottom portion of the attributes we now deal with the scrolling of the right hand side of the attributes.

First, we adjust the HL register pair to point to one row above the DE pointer. The instruction LD BC,−32 is the one used. The final portion of code is similar to the loop DOLEFT but this time we are only scrolling 20 rows as the second row from the top has its new attribute SAVEd on the stack.

```
        DEC   HL        ;RE-ADJUST HL BACK ONE
        LD    BC,-32    ;AND MAKE IT POINT TO ONE
        ADD   HL,BC     ;ROW ABOVE DE
        LD    B,20      ;THIS TIME ONLY DO 20 TIMES
                        ;AS LAST ATTRIBUTE IS HELD ON
                        ;THE STACK
DORIG:  LD    A,(HL)    ;GET THE ATTRIBUTE ABOVE
        LD    (DE),A    ;AND PLACE IT TO THE ATTRIBUTE
                        ;BELOW
        PUSH  BC        ;SAVE ROW COUNTER
        LD    BC,-32    ;LOAD BC WITH OFFSET
        ADD   HL,BC     ;MAKE HL POINT TO ONE ROW ABOVE
        EX    DE,HL     ;SAVE TEMP IN THE DE PAIR
        ADD   HL,BC     ;MAKE OLD DE POINT ONE ROW ABOVE
        EX    DE,HL     ;AND RESTORE BACK DE AND HL
        POP   BC        ;AS WELL AS THE ROW COUNTER
        DJNZ  DORIG     ;REPEAT UNTIL ALL ROWS DONE
```

Finally, we 'POP' off the first attribute that we saved and place it to

the last attribute on the second row of the screen. The RETurn instruction then RETURNS control back to the BASIC program.

```
        POP   AF        ;GET ATTRIBUTE VALUE ON STACK
        LD    (DE),A    ;AND PLACE IN NEW POSITION
        RET             ;AND RETURN

        END
```

```
        ORG     30000

ATTRADD EQU     5800H

        LD      HL,ATTRADD+31   ;POINT TO RIGHT HAND
                                ;SIDE OF ATTR
        LD      DE,ATTRADD+30   ;DE POINTS TO THE NEXT ATTR
        LD      A,(HL)          ;SAVE FIRST ATTRIBUTE
        PUSH    AF              ;ON THE STACK
        LD      B,31            ;LOAD B WITH COLUMN COUNTER
DOTOP:  LD      A,(DE)          ;GET NEXT ATTRIBUTE
        LD      (HL),A          ;AND PLACE IT TO THE
                                ;ATTRIBUTE TO THE LEFT
        DEC     DE              ;POINT TO THE NEXT
                                ;ATTRIBUTES
        DEC     HL
        DJNZ    DOTOP           ;REPEAT UNTIL DONE
                                ;ALL COLUMNS

;HL NOW POINTS TO 5800H IE TOP
;LEFT HAND CORNER OF ATTRIBUTES

        LD      BC,32           ;LET HL POINT ONE ROW DOWN
        ADD     HL,BC           ;BY ADDING 32
        INC     DE              ;DE NOW POINTS TO START
                                ;OF ATTRIBUTE FILE IE 5800H

        LD      B,21
```

```
DOLEFT: LD      A,(HL)          ;GET ATTRIBUTE BELOW
        LD      (DE),A          ;AND PLACE ON OLD ONE.
        PUSH    BC              ;SAVE ROW COUNTER
        LD      BC,32           ;NEXT ROW OFFSET
        ADD     HL,BC           ;HL NOW POINTS TO NEW ROW
        EX      DE,HL           ;SWAP FOR DE
        ADD     HL,BC           ;WHAT WAS DE NOW POINTS
                                ;TO NEW ROW
        EX      DE,HL           ;RESTORE BACK TO NORMAL
        POP     BC              ;RESTORE ROW COUNTER
        DJNZ    DOLEFT          ;REPEAT UNTIL DONE ALL ROWS


        LD      H,D             ;PLACE DE INTO HL
        LD      L,E
        INC     HL              ;HL POINTS ONE TO
                                ;RIGHT OF DE

        LD      B,31            ;LOAD B REG WITH COLUMN
                                ;COUNTER


DOBOT:  LD      A,(HL)          ;GET DATA FROM THE ATTR
                                ;ON RIGHT
        LD      (DE),A          ;AND PLACE IT IN  THE LEFT
        INC     HL              ;POINT TO NEW ATTRIBUTES
        INC     DE              ;TO THE RIGHT
        DJNZ    DOBOT           ;REPEAT UNTIL DONE 31 TIMES


        DEC     HL              ;RE-ADJUST HL BACK ONE
        LD      BC,-32          ;AND MAKE IT POINT TO ONE
        ADD     HL,BC           ;ROW ABOVE DE

        LD      B,20            ;THIS TIME ONLY DO 20 TIMES
                                ;AS LAST ATTRIBUTE
                                ;IS HELD ON THE STACK
```

```
DORIG:  LD      A,(HL)          ;GET THE ATTRIBUTE ABOVE
        LD      (DE),A          ;AND PLACE IT TO THE
                                ;ATTRIBUTE BELOW
        PUSH    BC              ;SAVE ROW COUNTER
        LD      BC,-32          ;LOAD BC WITH OFFSET
        ADD     HL,BC           ;MAKE HL POINT TO ONE
                                ;ROW ABOVE
        EX      DE,HL           ;SAVE TEMP IN THE DE PAIR
        ADD     HL,BC           ;MAKE OLD DE POINT ONE
                                ;ROW ABOVE
        EX      DE,HL           ;AND RESTORE BACK DE AND HL
        POP     BC              ;AS WELL AS THE ROW COUNTER
        DJNZ    DORIG           ;REPEAT UNTIL DONE ALL ROWS

        POP     AF              ;GET ATTRIBUTE VALUE
                                ;ON STACK
        LD      (DE),A          ;AND PLACE IN NEW POSTION
        RET                     ;AND RETURN
```

```
  5 CLEAR 29998:BORDER 0:CLS
 10 LET X=22528
 20 FOR F=X TO X+31
 30 POKE F,8*RND*7
 40 NEXT F
 50 FOR S=1 TO 21
 60 LET A=X+32*S
 70 POKE A,8*RND*7
 80 LET A=A+31
 90 POKE A,8*RND*7
100 NEXT S
110 FOR X=23201 TO 23201+30
130 POKE X,8*RND*7
140 NEXT X
150 RANDOMIZE USR 30000
160 PAUSE 10
170 GO TO 150
```

9 Using the ROM routines

The ROM (Read Only Memory) is a permanent program built into the Spectrum. It handles the interpretation and execution of BASIC and controls the operating system. It is contained in the first 16k of the Spectrum's memory map and enables you to RUN and edit BASIC programs. The ROM manages sound, graphics and communication between the cassette port and the keyboard.

It is worthwhile looking at some of the routines contained in the ROM as these can be used by the programmer when memory space is scarce. They can also be utilised if a complex arithmetic routine is needed or a particular function of the Spectrum is to be used. Each ROM routine is found at a particular memory location and we need to know the correct sequence of operations and the values that must be placed in selected registers.

PRINT and CHANNEL routines

The following routine allows the user to PRINT characters to what are known as *streams* via channels. A channel is the route by which input and output are effected to the various devices on a computer. Examples of such devices on the Spectrum are the keyboard, the printer and the screen. There are seven channels on the Spectrum, the most useful of which are given below:

Channel 0 or Channel K (used for input and output to bottom part of screen)
Channel 1 or Channel K (as Channel 0)
Channel 2 or Channel S (used for printing to the screen)
Channel 3 or Channel P (used for printing to the printer)

Before we PRINT to a channel we must indicate to the Spectrum which stream we wish to use. This is known as 'Opening a Channel'. When we use the ROM routines to PRINT a character the output will go to the currently selected channel until we open another channel. To open a channel we LOAD the A register with the channel number we wish to use. We then call the ROM routine at address 1601 hex which

will 'open' that channel. Any PRINTing done by means of the ROM routines would now send the output to the channel selected.

Therefore, if we wanted to start PRINTing to the screen we would open channel 2:

```
LD      A,2      ;select screen
CALL    1601H    ;open channel
```

Now to start PRINTing a character to the currently selected channel we can use a routine at the address 10 hex. The A register is LOADed with the ASCII value of the character we wish to PRINT. We use the call instruction:

```
RST     10H      ;print the character in A reg
                 ;to the current channel
```

The RST 10 subroutine is extremely useful when PRINTing to the screen because it automatically updates the system variables by updating the print co-ordinate of the next character position. It can also handle all the control characters, thus enabling us to simulate the PRINT AT ,TAB,INK,PAPER,OVER,INVERSE,BRIGHT and FLASH BASIC commands. Given below is a table of the control characters and their code values:

Character	ASCII value (decimal)
INK	16
PAPER	17
FLASH	18
BRIGHT	19
INVERSE	20
OVER	21
AT	22
TAB	23

Therefore if we wished to PRINT the character 'A' on the screen at Y position 10 and X position 10, in blue INK yellow PAPER we would use the code:

```
LD      A,2      ;first open channel 2
CALL    1601H
LD      A,22     ;PRINT AT
RST     10H
LD      A,10     ;PRINT AT10
RST     10
```

```
        LD      A,10
        RST     10H         ;PRINT AT 10,10;
        LD      A,16        ;INK
        RST     10H
        LD      A,1         ;LOAD A WITH CODE FOR BLUE
        RST     10H         ;BLUE INK
        LD      A,17        ;PAPER
        RST     10H
        LD      A,6         ;YELLOW PAPER
        LD      A,'A'
        RST     10          ;PRINT CHARACTER
```

PRINT STRING – 203C hex
This routine can be used to print a string of characters. The DE register pair is set up to contain the address of the start of the string that we wish to print, the BC register contains the number of characters in the string. As an example we have taken the last routine but this time all the characters have been put into a string:

```
        LD      A,2
        CALL    1601H       ;open channel 2
        LD      DE,string   ;point to string
        LD      BC,8        ;number of chars in string.
        CALL    203CH       ;print string.
        RET                 ;and return
string: DB      22,10,10,16,1,17,6,'A'
```

Printing numbers

PRINT LINE NUMBER – 1A1BH
This is a simple routine which is used by the ROM to PRINT line numbers in BASIC therefore it is limited to PRINTing numbers from 0 to 9999. The BC register pair is set up with the number we wish to PRINT;

```
        LD      BC,400H
        CALL    1A1BH       ;print 1024 decimal
        RET
```

PRINT LINE NUMBER2 – 1A28H
This routine is identical to the one above except that the number is pointed to by the HL pair.

```
LD      HL,6000H address   contents
CALL    1A28H    6000h     03H
RET              6001h     CAH
```

This section of code would result in the number 970 decimal being PRINTed to the current channel. You may notice that the two bytes that make up the number are stored in memory in the reverse manner to that usually used by the Z80 (i.e. MSB, LSB). This is due to the line numbers being held the 'wrong way' round in memory.

SCREEN ADDRESSING

Screen addressing routines in the ROM are used for PRINTing and PLOTting. When we PLOT a point using the Spectrum BASIC the screen is split into a grid of 176 lines by 256 points with the co-ordinate 0,0 starting at the bottom left hand side. There are 16 remaining lines at the bottom of the screen which are used by the BASIC operating system for INPUT, error messages, etc.

These routines may be used if required. However if you wish to PLOT or PRINT to any part of the screen, including the 16 'unusable' lines, I suggest you use my PLOT routine described in chapter 10 on the display file.

CHARADD – 0E9BH

This routine can be used to find the address on the screen for a given character line number 1–24. The first line starts at the bottom of the screen. The B register is LOADed with the line number of which we want to find the address. After calling the routine the HL register pair is RETURNed with the address of the first character line.

```
LD      B,1
CALL    0E9BH      ;find address of line 1
```

PIXADD – 22AAH

This routine is used to find the address of a point on the screen. The B register is set with Y co-ordinate and the C register with the X Co-ordinate, the address of which we wish to find. On RETURNing from the subroutine the HL register pair contains the screen address and the A register the bit position of the screen which is 0–7. Note that this is inverted and refers to the bit sequence left to right, not right to left. Therefore if A returns 0 then the leftmost bit, bit seven is referred to and if A gives 2 then bit five (the third from the left) is referred to, etc.

Beeper routine – 03B5

The BEEPer routine is called by LOADing the DE register with the duration, which is the frequency of the note multiplied by the number of seconds we wish the note to last. The pitch is LOADed into the HL register pair and we call the BEEPer routine at address 03B5 hex. The notes and their corresponding pitch values are given in the table below. For example, if we wanted to play middle G sharp for half a second (i.e. G#4) we would LOAD the DE register pair with 0.5*415=CF hex (415=19F hex), LOAD the HL pair with 3FF, and then call the routine:

```
;play middle G sharp
;for half a second
LD DE,0CFH
LD HL,19FH
CALL 03B5H
```

Table of duration and pitch values for musical notes

Note	Freq (Hz)	Dur (sec) (hex)	Pitch (hex)	Note	Freq (Hz)	Dur (sec) (hex)	Pitch (hex)
C 0	16.35	10	6868	A 1	55.00	37	1EF4
C#0	17.32	11	628D	A#1	58.28	3A	1D34
D 0	18.35	12	5D03	B 1	61.74	3D	1B90
D#0	19.44	13	57CB	C 2	65.40	41	1A03
E 0	20.60	14	52D7	C#2	69.20	45	188C
F 0	21.83	15	4E2B	D 2	73.40	49	172A
F#0	23.12	17	49CC	D#2	77.76	4D	15DC
G 0	24.50	18	45A3	E 2	82.40	52	149F
G#0	25.96	19	41B6	F 2	87.32	57	1374
A 0	27.50	1B	3E06	F#2	92.48	5C	125C
A#0	29.14	1D	3A87	G 2	98.00	62	1152
B 0	30.87	1E	373E	G#2	103.84	67	1057
C 1	32.70	20	3425	A 2	110.00	6E	F6B
C#1	34.64	22	3137	A#2	116.56	74	E8B
D 1	36.70	24	2E72	B 2	123.48	7B	DB8
D#1	38.88	26	2BD6	C 3	130.80	82	CF2
E 1	41.20	29	295C	C#3	138.56	8A	C37
F 1	43.66	2B	2706	D 3	146.80	92	B86
F#1	46.24	2E	24D7	D#3	155.52	9B	ADF
G 1	49.00	31	22C2	E 3	164.80	A4	A40
G#1	51.92	33	20CC	F 3	174.64	AE	9AB

Note	Freq (Hz)	Dur (sec) (hex)	Pitch (hex)	Note	Freq (Hz)	Dur (sec) (hex)	Pitch (hex)
F#3	184.96	B8	91F	A 5	880.00	370	1D3
G 3	196.00	C4	89A	A#5	932.48	3A4	1B7
G#3	207.68	CF	81C	B 5	987.84	3DB	19C
A 3	220.00	DC	7A6	C 6	1046.40	416	183
A#3	233.12	E9	736	C#6	1108.48	454	16C
B 3	246.96	F6	6CD	D 6	1174.40	496	156
C 4	261.60	105	66A	D#6	1244.16	4DC	141
C#4	277.12	115	60C	E 6	1318.40	526	12D
D 4	293.60	125	5B3	F 6	1397.12	575	11B
D#4	311.04	137	560	F#6	1479.68	5C7	109
E 4	329.60	149	511	G 6	1568.00	620	F8
F 4	349.28	15D	4C6	G#6	1661.44	67D	E9
F#4	369.92	171	480	A 6	1760.00	6E0	DA
G 4	392.00	188	43D	A#6	1864.96	748	CC
G#4	415.36	19F	3FF	B 6	1975.68	7B7	BF
A 4	440.00	1B8	3C4	C 7	2092.80	82C	B2
A#4	466.24	1D2	38C	C#7	2216.96	8A8	A7
B 4	493.92	1ED	357	D 7	2348.80	92C	9C
C 5	523.20	20B	326	D#7	2488.32	9B8	91
C#5	554.24	22A	2F7	E 7	2636.80	A4C	87
D 5	587.20	24B	2CA	F 7	2794.24	AEA	7E
D#5	622.08	26E	2A1	F#7	2959.36	B8F	75
E 5	659.20	293	279	G 7	3136.00	C40	6D
F 5	698.56	2BA	254	G#7	3322.88	CFA	65
F#5	739.84	2E3	231	A 7	3520.00	DC0	5E
G 5	784.00	310	20F	A#7	3729.92	E91	57
G#5	830.72	33E	1F0	B 7	3951.36	F6F	50

As a demonstration the program below plays a well known tune which true Spectrum owners should recognise. The program reads a series of pitch and duration values which are passed to the BEEPer. To finish the tune we use the byte 0FF hex. If you are really into music a more sophisticated version of PLAY is given in Chapter 12.

Assembler Listing

```
        ORG     32000D
BEEPER  EQU     03B5H

PLAY:
        LD      IX,FRERE        ;POINT TO START OF MUSIC
GETV:   LD      L,(IX+0)        ;LOAD L WITH LOW PART
                                ;OF THE PITCH
        LD      H,(IX+1)        ;LOAD H WITH HIGH PART
                                ;OF THE PITCH
        INC     H               ;IF H IS 0FF HEX THEN
                                ;END OF MUSIC
        RET     Z               ;SO RETURN
        DEC     H               ;RESTORE HIGH PART OF PITCH
        LD      E,(IX+2)        ;LOAD E WITH LOW PART
                                ;OF DURATION
        LD      D,(IX+3)        ;LOAD D WITH HIGH PART
                                ;OF DURATION
        PUSH    IX              ;SAVE MUSIC POINTER ON
                                ;THE STACK
        CALL    BEEPER          ;CALL BEEPER ROUTINE IN
                                ;THE ROM
        POP     IX              ;RESTORE IX
        LD      DE,4            ;LOAD DE WITH 4
        ADD     IX,DE           ;AND POINT TO NEXT
                                ;PIECE OF MUSIC
        JR      GETV            ;GET NEXT PIECE OF MUSIC

FRERE:

        DEFW    66AH
        DEFW    105H
        DEFW    5B3H
        DEFW    125H
        DEFW    560H
        DEFW    9BH
        DEFW    5B3H
        DEFW    92H
        DEFW    66AH
        DEFW    105H
```

```
DEFW    66AH
DEFW    105H
DEFW    5B3H
DEFW    125H
DEFW    560H
DEFW    9BH
DEFW    5B3H
DEFW    92H
DEFW    66AH
DEFW    105H


DEFW    560H
DEFW    137H
DEFW    4C6H
DEFW    15DH
DEFW    43DH
DEFW    188H


DEFW    560H
DEFW    137H
DEFW    4C6H
DEFW    15DH
DEFW    43DH
DEFW    188H

DEFW    43DH
DEFW    126H
DEFW    3FFH
DEFW    67H
DEFW    43DH
DEFW    0C4H
DEFW    4C6H
DEFW    0AEH
DEFW    560H
DEFW    9BH
DEFW    5B3H
DEFW    92H
DEFW    66AH
DEFW    105H
```

```
DEFW    43DH
DEFW    126H
DEFW    3FFH
DEFW    67H
DEFW    43DH
DEFW    0C4H
DEFW    4C6H
DEFW    0AEH
DEFW    560H
DEFW    9BH
DEFW    5B3H
DEFW    92H
DEFW    66AH
DEFW    105H

DEFW    66AH
DEFW    105H
DEFW    89AH
DEFW    0C4H
DEFW    66AH
DEFW    20AH

DEFW    66AH
DEFW    105H
DEFW    89AH
DEFW    0C4H
DEFW    66AH
DEFW    20AH
DEFW    0FFFFH
END
```

Hexadecimal Listing

```
7D00    DD  21  21  7D  DD  6E  00  DD
7D08    66  01  24  C8  25  DD  5E  02
7D10    DD  56  03  DD  E5  CD  B5  03
7D18    DD  E1  11  04  00  DD  19  18
7D20    E3  6A  06  05  01  B3  05  25
7D28    01  60  05  9B  00  B3  05  92
7D30    00  6A  06  05  01  6A  06  05
7D38    01  B3  05  25  01  60  05  9B
```

```
7D40   00  B3  05  92  00  6A  06  05
7D48   01  60  05  37  01  C6  04  5D
7D50   01  3D  04  88  01  60  05  37
7D58   01  C6  04  5D  01  3D  04  88
7D60   01  3D  04  26  01  FF  03  67
7D68   00  3D  04  C4  00  C6  04  AE
7D70   00  60  05  9B  00  B3  05  92
7D78   00  6A  06  05  01  3D  04  26

7D80   01  FF  03  67  00  3D  04  C4
7D88   00  C6  04  AE  00  60  05  9B
7D90   00  B3  05  92  00  6A  06  05
7D98   01  6A  06  05  01  9A  08  C4
7DA0   00  6A  06  0A  02  6A  06  05
7DA8   01  9A  08  C4  00  6A  06  0A
7DB0   02  FF  FF  05  01  6A  06  05
7DB8   01  B3  05  25  01  60  05  9B

7DC0   00  B3  05  92  00  6A  06  05
7DC8   01  60  05  37  01  C6  04  5D
7DD0   01  3D  04  88  01  60  05  37
7DD8   01  C6  04  5D  01  3D  04  88
7DE0   01  3D  04  26  01  FF  03  67
7DE8   00  3D  04  C4  00  C6  04  AE
7DF0   00  60  05  9B  00  B3  05  92
7DF8   00  6A  06  05  01  3D  04  26
```

TAPE LOADING AND SAVING

The following routines can be used in programs to enable the user to SAVE and LOAD data to cassette. When you SAVE or LOAD data in BASIC it comes in two parts. The first part is a 'header' containing information about the file, its name, the length of the file, and its start address. The two main routines are SAVEDATA, at address 04C2 hex, and LOADDATA at 0556 hex. These routines are called with the IX register pair containing the address where the data is placed or SAVEd from and the DE register containing the number of bytes we wish to SAVE or LOAD. We LOAD the A register with 0 if we are dealing with the header section or 0FF hex if we are dealing with a data block. When LOADing a file the Carry flag is set when actually LOADing data but it is reset if we want to VERIFY data.

Below is a set of routines which the programmer can use with the cassette.

```
SAVEBYTES:
LD      DE,NBYTES                ;number of bytes to save
LD      IX,START                 ;start of block
LD      A,0FFH                   ;saving a data block
CALL    SAVEDATA                 ;save bytes
RET

SAVEHEADER:
LD      DE,17                    ;save 17 bytes length of header
LD      IX,START-OF-HEADER       ;point to start of header info
XOR     A                        ;signify header.A=0
CALL    SAVEDATA                 ;save header
RET

LOADBYTES:
LD      DE,NBYTES
LD      IX,START
LD      A,0FFH
SCF                              ;signify loading
CALL    LOADDATA
RET

LOADHEADER:
LD      DE,17
LD      IX,START-of-HEADER
XOR     A
SCF
CALL    LOADDATA
RET

VERIFYBYTES:
LD      DE,NBYTES
LD      IX,START
LD      A,0FFH
AND     A                        ;reset carry flag
CALL    LOADDATA
RET

VERIFYHEADER:
LD      DE,17
LD      IX,START-of-HEADER
XOR     A                        ;reset carry flag and
CALL    LOADDATA                 ;set A=0
RET
```

The following program uses these routines to make backup copies of most tapes. Please do not infringe copyright by using this to copy protected tapes!

Assembler Listing

```
        ORG     32000
        JP      START


MEM     EQU     23900
PRXSTRING       EQU     203CH




MESS1:  DEFM    22,1,0,'FILENAME: '
SIZ1    EQU     13
MESS2:  DEFM    22,3,0,'PROGRAM TYPE: '
SIZ2    EQU     17
MESS3:  DEFM    22,6,0,'LENGTH: '
SIZ3    EQU     11
MESS4:  DEFM    22,9,0,'START: '
SIZ4    EQU     10
;MESS5: DEFM    22,12,0,'BASIC LENGTH:'
;SIZ5   EQU     16
WAITP:  DEFM    22,10,0,'     MAC WAITING FOR HEADER. '
        DEFM    22,13,0,' COPYRIGHT J.K WILSON 1983. '
SIZWP   EQU     62

TBASIC: DEFM    'BASIC'
TNUM:   DEFM    'NUMER'
TCHAR:  DEFM    'CHARA'
TCODE:  DEFM    'BYTES'

SAVEQ:  DEFM    22,21,0, 'Do you want a copy?'
SIZQ    EQU     22
SAVER:  DEFM    22,21,0,'Press ENTER when ready.'
SIZR    EQU     26
BLANKM: DEFM    22,21,0,'                        '
SIZB    EQU     26
```

```
ERRXSP   EQU    23613          ;SYSTEM VARIABLE
                               ;ERROR STACK POINTER
LOADBTS  EQU    0556H          ;LOAD BYTES ROM ROUTINE
SAVEBTS  EQU    04C2H          ;SAVE BYTES ROM ROUTINE

NUMTA:                         ;BASE TEN TABLE
         DEFW   10000
         DEFW   1000
         DEFW   100
         DEFW   10
         DEFW   1

NUMB:    DS     5              ;NUMBER BUFFER

OUTXNUM:
         LD     IX,NUMTA       ;POINT TO TABLE
         LD     DE,NUMB        ;DE POINTS TO BUFFER
DIGIT:   LD     C,(IX+0)       ;GET LOW BYTE OF BASE 10
         LD     B,(IX+1)       ;GET HIGH BYTE OF BASE 10
         LD     A,'0'-1        ;A REGISTER =30 HEX
         AND    A              ;CLEAR CARRY
FIN:
         INC    A              ;CALCULATE NUMBER OF
         SBC    HL,BC          ;MULTIPLES OF TENS UNTIL
         JR     NC,FIN         ;CARRY FLAG IS SET
         ADD    HL,BC          ;RESTORE NUMBER
         LD     (DE),A         ;PLACE ASCII NUMBER
                               ;IN BUFFER
         DEC    C              ;TEST TO SEE IF FINISED
         INC    DE             ;BUMP BUFFER POINTER
         JR     Z,OUTP         ;FINISHED OUTPUT NUMBER
                               ;TO CURRENT CHANNEL
         INC    IX             ;POINT TO NEXT
                               ;MULTIPLE OF 10
         INC    IX
         JR     DIGIT          ;FIND NEXT ASCII DIGIT
OUTP:    LD     DE,NUMB        ;NUMBER BUFFER
         LD     BC,5           ;LENGTH OF STRING
         CALL   PRXSTRING      ;PRINT IT!
         RET                   ;AND RETURN
```

```
;HEAD GETS HEAD INFORMATION FROM TAPE
;17 BYTES OF INFORMATION ARE PASSED TO STARTING ADDRESS IX
;NB. DE MUST BE LOADED WITH 17
;
;0-1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16
;-----------------------------------------
;T|*****FILENAME*******|*LEN*|*STR*|*PRG*|
;-----------------------------------------
;T=TYPE 0 :BASIC PROGRAM
;       1 :NUMERICAL ARRAY
;       2 :CHARACTER ARRAY
;       3 :BLOCK OF CODE
;
;FILENAME :NO MORE THAN TEN BYTES
;
;LEN=LENGTH OF CODE IF TYPE 3
;
;STR=START ADDRESS OF CODE OR LINE NUMBER
;
;PRG=LENGTH OF PROGRAM AREA
;
HEADER: DS      17

TYP     EQU     HEADER+0        ;TYPE OF PROGRAM
FILE    EQU     HEADER+1        ;FILENAME
LEN     EQU     HEADER+11       ;LENGTH OF CODE
STR     EQU     HEADER+13       ;STARTING ADDRESS

HEADIN: SCF                     ;SET CARRY FLAG
        LD      A,0
        LD      IX,HEADER       ;IX POINTS TO HEADER BUFFER
        LD      DE,17           ;17 BYTES OF INFORMATION
        CALL    LOADBTS         ;LOAD BYTES
        RET


HEADOUT:
        LD      A,0             ;SET A TO ZERO
        LD      IX,HEADER       ;POINT TO HEADER
        LD      DE,17           ;17 BYTES TO
        CALL    SAVEBTS         ;SAVE
        RET
```

```
SAVECODE:
        LD      A,0FFH
        LD      IX,MEM          ;POINT TO RAM
        LD      DE,(LEN)        ;GET LENGTH FROM HEADER
        CALL    SAVEBTS         ;AND SAVE
        RET



LOADCODE:
        SCF                     ;SET CARRY FLAG
                                ;TO SIGNIFY LOADING
        LD      A,0FFH          ;A REG LOADED WITH
                                ;TYPE OF DATA
        LD      IX,MEM          ;POINT TO START OF CODE
        LD      DE,(LEN)        ;PUT LENGTH OF CODE INTO DE
        CALL    LOADBTS         ;DO LOADING
        RET

CLS:
        LD      HL,4000H        ;CLEAR SCREEN
        LD      DE,4001H
        LD      BC,8*32*24-1
        LD      (HL),0
        LDIR
        LD      DE,HOME         ;AND PLACE CURSOR
        LD      BC,3
        CALL    PRXSTRING       ;AT HOME
        RET

HOME:   DB      22,0,0


DISPLAY:
        CALL    CLS             ;CLEAR SCREEN
        LD      DE,MESS1        ;PRINT FILENAME STRING
        LD      BC,SIZ1
        CALL    PRXSTRING
        LD      DE,FILE
        LD      BC,10           ;PRINT
        CALL    PRXSTRING       ;FILENAME
```

```
        LD      DE,MESS2        ;PRINT TYPE STRING
        LD      BC,SIZ2
        CALL    PRXSTRING

        LD      HL,TBASIC
        LD      A,(TYP)
        LD      E,A             ;SAVE TYP IN E REG
        SLA     A               ;2*TY
        SLA     A               ;4*TYP
        ADD     A,E             ;4*TYP+TYP=5*TYP
        LD      E,A
        LD      D,0             ;PUT OFFSET IN DE
        ADD     HL,DE
        ;HL POINTS TO STRING
        EX      DE,HL
        LD      BC,5            ;NUMBER OF BYTES TO PRINT
        CALL    PRXSTRING       ;TYPE

        LD      DE,MESS3        ;PRINT LENGTH STRING
        LD      BC,SIZ3
        CALL    PRXSTRING
        LD      HL,(LEN)
        CALL    OUTXNUM         ;NUMBER OF BYTES



        LD      DE,MESS4        ;STARTING LINE/ADDRESS
        LD      BC,SIZ4
        CALL    PRXSTRING
        LD      HL,(STR)
        CALL    OUTXNUM
        RET


START:
        LD      SP,STACK
        LD      A,2             ;OPEN CHANNEL 'S'
        CALL    1601H
```

```
ERRORS:
        LD      SP,STACK
        LD      HL,ERRSP        ;ERROR STACK
        LD      (HL),LOW(ERRORS)
        INC     HL
        LD      (HL),HIGH(ERRORS)
        DEC     HL
        LD      (23613),HL

NEXT:
        CALL    WAITM           ;WAIT FOR HEADER MESSAGE

        CALL    HEADIN          ;GET HEADER
        CALL    DISPLAY         ;DISPLAY INFORMATION
        CALL    LOADCODE        ;LOAD CODE
        CALL    WANT            ;DOES HE WANT TO SAVE THIS?
        JR      NZ,NEXT
ACOPY:  CALL    SAVEMESS        ;ASK HIM IF HE IS READY
                                ;TO SAVE ETC ETC.
        CALL    HEADOUT         ;OUTPUT HEADER
        CALL    PAUSE           ;WAIT
        CALL    PAUSE
        CALL    SAVECODE        ;SAVE CODE
        CALL    WANT            ;DOES HE WANT TO MAKE
                                ;ANOTHER COPY?
        JR      Z,ACOPY         ;YES.WELL MAKE ANOTHER ONE
        JR      NEXT
WAITM:
        CALL    CLS             ;CLEAR SCREEN
        LD      DE,WAITP        ;TELL THEM
        LD      BC,SIZWP        ;WE ARE WAITING
        CALL    PRXSTRING
        RET

WANT:
        LD      DE,SAVEQ        ;PROMPT FOR ANSWER
        LD      BC,SIZQ
        CALL    PRXSTRING
```

```
WAITK:  CALL    KEY             ;GET KEYBOARD STATUS
        CP      'Y'
        JR      Z,RSZ           ;IF YES RETURN WITH A
        CP      'y'
        JR      Z,RSZ           ;ZERO
        CP      'N'
        JR      Z,RSZ-1         ;NON ZERO
        CP      'n'
        JR      NZ,WAITK
        AND     A               ;NON ZERO FLAG RESET
RSZ:    PUSH    AF              ;SAVE FLAGS
BLK:    LD      DE,BLANKM       ;BLANK OUT BOTTOM MESSAGE
        LD      BC,SIZB
        CALL    PRXSTRING
        POP     AF              ;RESTORE FLAGS
        RET
PAUSE:
        CALL    PAUSE2          ;WAIT A WHILE
PAUSE2: LD      HL,0
        LD      DE,0
        LD      BC,0FFFFH
        LDIR
        RET

SAVEMESS:
        LD      BC,SIZR         ;PRINT MESSAGE
        LD      DE,SAVER        ;TO PROMPT FOR ENTER
        CALL    PRXSTRING
ENT:    CALL    KEY
        CP      0DH
        JR      NZ,ENT          ;WAIT FOR ENTER
        CALL    RSZ             ;BLANK OUT BOTTOM SCREEN
        RET

FLAGS   EQU     23611           ;STATE OF KEYBOARD
LASTXK  EQU     23560           ;LAST KEY PRESSED
```

```
KEY:
DEPR:
        LD      A,(FLAGS)       ;LOOK AT STATUS OF KEYBOARD
        BIT     5,A
        JR      Z,KEY           ;WAIT FOR A KEY
                                ;TO BE PRESSED
        RES     5,A
        LD      (FLAGS),A
        LD      A,(LASTXK)      ;GET KEY VALUE
        RET
        DS      100             ;STACK SPACE
STACK:  DB      0
ERRSP:  DEFW    0               ;ERROR STACK SPACE
```

Hexadecimal Listing

```
7D00   C3  C3  7E  16  01  00  46  49
7D08   4C  45  4E  41  4D  45  3A  20
7D10   16  03  00  50  52  4F  47  52
7D18   41  4D  20  54  59  50  45  3A
7D20   20  16  06  00  4C  45  4E  47
7D28   54  48  3A  20  16  09  00  53
7D30   54  41  52  54  3A  20  16  0A
7D38   00  20  20  20  20  4D  41  43

7D40   20  57  41  49  54  49  4E  47
7D48   20  46  4F  52  20  48  45  41
7D50   44  45  52  2E  20  16  0D  00
7D58   20  43  4F  50  59  52  49  47
7D60   48  54  20  4A  2E  4B  20  57
7D68   49  4C  53  4F  4E  20  31  39
7D70   38  33  2E  20  42  41  53  49
7D78   43  4E  55  4D  45  52  43  48
```

7D80	41	52	41	42	59	54	45	53
7D88	16	15	00	44	6F	20	79	6F
7D90	75	20	77	61	6E	74	20	61
7D98	20	63	6F	70	79	3F	16	15
7DA0	00	50	72	65	73	73	20	45
7DA8	4E	54	45	52	20	77	68	65
7DB0	6E	20	72	65	61	64	79	2E
7DB8	16	15	00	20	20	20	20	20
7DC0	20	20	20	20	20	20	20	20
7DC8	20	20	20	20	20	20	20	20
7DD0	20	20	10	27	E8	03	64	00
7DD8	0A	00	01	00	00	00	00	00
7DE0	00	DD	21	D2	7D	11	DC	7D
7DE8	DD	4E	00	DD	46	01	3E	2F
7DF0	A7	3C	ED	42	30	FB	09	12
7DF8	0D	13	28	06	DD	23	DD	23
7E00	18	E6	11	DC	7D	01	05	00
7E08	CD	3C	20	C9	00	00	00	00
7E10	00	00	00	00	00	00	00	00
7E18	00	00	00	00	00	37	3E	00
7E20	DD	21	0C	7E	11	11	00	CD
7E28	56	05	C9	3E	00	DD	21	0C
7E30	7E	11	11	00	CD	C2	04	C9
7E38	3E	FF	DD	21	5C	5D	ED	5B
7E40	17	7E	CD	C2	04	C9	37	3E
7E48	FF	DD	21	5C	5D	ED	5B	17
7E50	7E	CD	56	05	C9	21	00	40
7E58	11	01	40	01	FF	17	36	00
7E60	ED	B0	11	6C	7E	01	03	00
7E68	CD	3C	20	C9	16	00	00	CD
7E70	55	7E	11	03	7D	01	0D	00
7E78	CD	3C	20	11	0D	7E	01	0A
7E80	00	CD	3C	20	11	10	7D	01
7E88	11	00	CD	3C	20	21	74	7D
7E90	3A	0C	7E	5F	CB	27	CB	27
7E98	83	5F	16	00	19	EB	01	05
7EA0	00	CD	3C	20	11	21	7D	01
7EA8	0B	00	CD	3C	20	2A	17	7E
7EB0	CD	E1	7D	11	2C	7D	01	0A
7EB8	00	CD	3C	20	2A	19	7E	CD

7EC0	E1	7D	C9	31	CF	7F	3E	02
7EC8	CD	01	16	31	CF	7F	21	D0
7ED0	7F	36	CB	23	36	7E	2B	22
7ED8	3D	5C	CD	01	7F	CD	1D	7E
7EE0	CD	6F	7E	CD	46	7E	CD	0E
7EE8	7F	20	EF	CD	46	7F	CD	2B
7EF0	7E	CD	37	7F	CD	37	7F	CD
7EF8	38	7E	CD	0E	7F	28	EC	18
7F00	D9	CD	55	7E	11	36	7D	01
7F08	3E	00	CD	3C	20	C9	11	88
7F10	7D	01	16	00	CD	3C	20	CD
7F18	5A	7F	FE	59	28	0D	FE	79
7F20	28	09	FE	4E	28	04	FE	6E
7F28	20	ED	A7	F5	11	B8	7D	01
7F30	1A	00	CD	3C	20	F1	C9	CD
7F38	3A	7F	21	00	00	11	00	00
7F40	01	FF	FF	ED	B0	C9	01	1A
7F48	00	11	9E	7D	CD	3C	20	CD
7F50	5A	7F	FE	0D	20	F9	CD	2B
7F58	7F	C9	00	3A	3B	5C	CB	6F
7F60	28	F8	CB	AF	32	3B	5C	3A
7F68	08	5C	C9	00	00	00	00	00
7F70	00	00	00	00	00	00	00	00
7F78	00	00	00	00	00	00	00	00
7F80	00	00	00	00	00	00	00	00
7F88	00	00	00	00	00	00	00	00
7F90	00	00	00	00	00	00	00	00
7F98	00	00	00	00	00	00	00	00
7FA0	00	00	00	00	00	00	00	00
7FA8	00	00	00	00	00	00	00	00
7FB0	00	00	00	00	00	00	00	00
7FB8	00	00	00	00	00	00	00	00
7FC0	00	00	00	00	00	00	00	00
7FC8	00	00	00	00	00	00	00	00
7FD0	00	00	FE	0D	20	F9	CD	2B
7FD8	7F	C9	00	3A	3B	5C	CB	6F
7FE0	28	F8	CB	AF	32	3B	5C	3A
7FE8	08	5C	C9	00	00	00	00	00
7FF0	00	00	00	00	00	00	00	00
7FF8	00	00	00	00	00	00	00	00

FLOATING POINT CALCULATION ROUTINES

The floating point calculator is used by the BASIC interpreter to handle calculations on floating point numbers and strings. Numbers in BASIC are represented by 5 bytes. The way these bytes represent a floating point number is shown in the Spectrum user's manual (chapter 24).

When the Spectrum is interpreting BASIC calculations it uses something known as a calculator stack which stores the interpreted calculation in a series of numbers, strings and operators. The way this line is interpreted is known as REVERSE POLISH NOTATION or RPN.

Numeric expressions in BASIC consist of operands, variables and operators or functions. In an algebraic expression operators are placed between two operands. For example, in the expression X+Y the operator is +, and the operands are X and Y. The principle of RPN is to place the operator after the operands. Thus X+Y is written as X Y +.

the larger algebraic expression:

A*B+C*D/2

could be written as:

A B * C D 2 / * +

To READ a RPN expression we use the following technique. READing the expression from left to right any values or operands would be pushed onto a stack. When we arrive at an operand or function the appropriate number of data is POPped off the stack and that operation or function is executed with the data. The 'new' value is then pushed onto the stack. This is repeated until we have reached the end of the expression:

2 4 * 3 8 2 / * +

For example, for the above RPN expression we go through the operations shown below. On the lefthand side of the diagram I have given the operations we use as we read the expression from left to right. On the righthand side the current status of the stack is shown. The top of the stack is the rightmost digit.

Operation	Stack
stack 2	2
stack 4	2 4
operator *	8
stack 3	8 3
stack 8	8 3 8
stack 2	8 3 8 2
operator /	8 3 4
operator *	8 12
operator +	20

The answer left on the stack is 20.

The Spectrum has a calculator stack where numbers can be manipulated in the same way as we would deal with an RPN expression. Groups of numbers can be pushed onto a stack and routines can be called to do various operations such as add, subtract and multiply. We can push numbers onto the stack by calling routines within the ROM which allow us to SAVE the numbers in 1 byte, 2 byte or 5 byte form.

```
STACKA–2D28H
LD        A,20
CALL      2D28H      ;STACK NUMBER 20
```

This routine will convert the one byte number contained in the Accumulator to its five byte floating format which is then pushed onto the calculator stack.

There are two other routines which are similar to the last routine. They allow us to stack a two byte integer in the BC register pair and a five byte floating point number contained in the A,E,D,C,B registers. The routine at the address 2D2B hex (STACKBC) will convert a two byte integer into five byte floating point format and push this on the calculator stack. Likewise the routine at the address 2AB6 hex (STACK5) will push the floating point number in the registers A,E,D,C and B.

To retrieve numbers from the stack we have routines which can pop them off and convert them into one byte, two byte or the normal five byte form. The addresses are given below:

UNSTACKA –2D5DH will convert the floating 5 byte number on top of the calculator stack to its equivalent one byte integer and place it in the accumulator.

UNSTACKBC –2D2AH will remove a five byte floating point number from the stack, convert it and place it in the BC register pair.

UNSTACK5 – 2BF1H. This routine is used to place a floating point number from the stack into the registers A,E,D,C and B.

There are two routines which are very useful to the programmer when handling floating point numbers. The routine at the address 2DE3 hex (PRINTFP) will take the top number on the calculator stack and PRINT it to the current channel selected. The second routine at 2C9B hex (ASCTOFP) enables us to convert a number from a string to a floating point number which is pushed to the calculator stack. Look at the following program:

```
LD      HL,STRING   ;put to start of string.
LD      (5C5DH),HL  ;save in system variable CH–ADD
LD      A,(HL)      ;get first character in A reg.
CALL    2C9BH       ;convert ascii number to fp.
CALL    2DE3H       ;print fp number
RET
DEFM    2.31693
DB      0DH         ;carriage return.
```

The HL register pair is set up to point to the start of the string. This is stored in the system variable CH–ADD which usually holds the address of the next character to be interpreted when RUNning Spectrum BASIC. The A register is LOADed with the first character and the routine ASCTOFP is called. This leaves the binary floating point number at the top of the calculator stack so when we call the routine PRINTFP (2DE3H) the number 2.31693 will be printed out. The end of the string is signified by a carriage return.

To start the floating point calculator we call the routine at address 28 hex with the one byte instruction RST 28H. The data following the call instruction indicates which operations the calculator must perform. The calculator goes through each operation automatically pushing and popping data until it reaches the data 38 hex which signifies the end of the calculation. Some of the most useful codes are given below:

data code (hex)	Name	Action
01	exchange	Swops the two topmost floating numbers.
02	delete	Deletes top number on stack.
03	subtract	Subtracts second number on stack from first
04	multiply	Multiplies the two topmost numbers.
05	divide	Divides the first over the second number.

data code (hex)	Name	Action
06	power	Raises the first to the power of the second
1B	negate	Changes the sign of the top number.
1F	sin	Calculates the sine of the top number.
20	cos	Calculates the cosine of the top number.
21	tan	Calculates the tangent of the top number.
22	arcsin	Calculates the Arcsine of the top number.
23	arccos	Calculates the Arccosine of the top number.
24	arctan	Calculates the Arctan of the top number.
25	log2	Calculates the Log of the top number.
26	exp	Calculates the exponential of the top number.
27	integer	Calculates the integer of the top number.
28	square root	Calculates the square root of the top number.
29	sign	Places the sign of the top number on stack
2A	absolute	Converts the top number to its absolute.
2B	peek	Places the contents of the address at top of stack
2C	in	Scans address at top of stack.
31	duplicate	Duplicate the top of the stack.
38	Endcalc	End of calculation.

The calculator has five constants available in the ROM used for calculating sines and cosines:

Data code (hex)	Name	Action
A0	stack 0	Place the number 0 on the stack.
A1	stack 1	Place the number 1 on the stack.
A2	stack 1/2	Place the number 1/2 on the stack.
A3	stack PI/2	Place half of PI on the stack.
A4	stack 10	Place the number 10 on the stack.

There are six memory locations used by the floating point calculator in order to SAVE numbers on top of the stack. The data codes C0 hex to C5 hex are used to SAVE the topmost number on the stack to one of the six memory locations. The other codes E0 hex to E5 hex are used to place numbers from one of the memory locations to the top of the calculator stack.

Data code (hex)	Name	Action
C0	store 0	Place the number on the stack in memory 0.
C1	store 1	Place the number on the stack in memory 1.
C2	store 2	Place the number on the stack in memory 2.
C3	store 3	Place the number on the stack in memory 3.
C4	store 4	Place the number on the stack in memory 4.
C5	store 5	Place the number on the stack in memory 5.
E0	Stack mem 0	Place the contents of memory 0 on the stack.
E1	Stack mem 1	Place the contents of memory 1 on the stack.
E2	Stack mem 2	Place the contents of memory 2 on the stack.
E3	Stack mem 3	Place the contents of memory 3 on the stack.
E4	Stack mem 4	Place the contents of memory 4 on the stack.
E5	Stack mem 5	Place the contents of memory 5 on the stack.

The following program demonstrates how we use the floating calculator to multiply the numbers 2342 (926 hex) and 156 (9C hex).

```
LD      BC,926H
CALL    STACKBC     ;STACK NUMBER
LD      BC,9CH
CALL    STACKBC     ;STACK NUMBER
RST     28H         ;START CALCULATION.
DB      04          ;MULTIPLY
DB      38H         ;END OF CALC.
CALL    PRINTFP     ;PRINT ANSWER
RET
```

The second example shows us how we can convert fairly complex algorithms into machine code. It PLOTS a sine wave on the screen. The BASIC routine takes 17 seconds while the machine code equivalent takes 14 seconds. The reason why the machine code routine is not significantly faster is because the sine calculations inside the ROM are fairly slow.

```
10 FOR X=0 TO 255
20 LET Y=100+50*SIN(X*PI/20)
30 PLOT X,Y
40 NEXT X
```

The equivalent machine code program would be:

```
        ORG     0


PLOTXY  EQU     22EEH
STACKA  EQU     2D28H
UNSTACKA        EQU     2DD5H


        XOR     A                   ;SET X CO-ORD TO 0
SINE:   PUSH    AF                  ;SAVE X CO-ORD ON STACK
        LD      A,100               ;STACK 100 DECIMAL
        CALL    STACKA
        LD      A,50
        CALL    STACKA              ;STACK 50 DECIMAL
        POP     AF                  ;GET X CO-ORD
        PUSH    AF                  ;AND SAVE
        CALL    STACKA              ;PLACE X ON FP STACK
        RST     28H                 ;START CALCULATION
        DB      0A3H                ;STACK PI/2
        DB      0A4H                ;STACK 10 DECIMAL
        DB      05H                 ;DIVIDE (PI/2 BY 10)
        DB      04H                 ;MULTPLY (PI/20 BY X)
        DB      1FH                 ;SINE (SINE(PI*X/20))
        DB      04H                 ;MULTIPLY
                                    ;(50*SIN(PI*X/20))
        DB      0FH                 ;ADD

                                    ;100+50*SIN(PI*X/20)
        DB      38H                 ;END OF CALCULATION

        CALL    UNSTACKA            ;GET Y CO-ORD IN A REG
        LD      B,A                 ;AND SAVE IN B

        POP     AF                  ;GET X CO-ORD OFF STACK
        PUSH    AF                  ;SAVE AGAIN

        LD      C,A                 ;PLACE X CO-ORD IN C REG
        CALL    PLOTXY              ;PLOT A POINT AT X,Y
        POP     AF                  ;GET X REGISTER
        INC     A                   ;INCREASE X CO-ORD
        RET     Z                   ;DONE 256 TIMES SO RETURN
        JR      SINE                ;KEEP PLOTTING

        END
```

10 Screen and attribute handling

The screen display is used in order to communicate information to the user. To PRINT information to the screen is a fairly simple and trivial matter in BASIC. However, when we delve into the realms of machine code accessing the screen becomes more complicated due to the complex screen lay out. The display file is split into two sections; the actual screen display file and the attribute file. The attribute file is easy to access because the data held in this file is organised sequentially. The attribute file consists of a 32 by 24 byte array. Each byte contains the information relating to a particular character position on the screen and tells the Spectrum what PAPER and INK colours to use for the display and whether the character is FLASH and/or BRIGHT.

The Attribute File

The number to be stored in the attribute file can be calculated by using the following method:

First set your total to zero.
Add 128 to your total if you want it flashing.
Add 64 to your total if you want it bright.
Add 8 times the PAPER colour you want.
Add the INK colour you want.

The value of the colours are:

0 Black
1 Blue
2 Red
3 Magenta
4 Green
5 Cyan
6 Yellow
7 White

The bit pattern of an attribute byte is set up like this:

Bit	7	6	5	4	3	2	1	0
	f	b	p	p	p	i	i	i

Where f is the FLASH bit, b is the BRIGHT bit, p is PAPER number, i is INK number. So if, for example, we wanted the colour code for red INK on white PAPER with the BRIGHTness set on we would put the value 64+8*7+2=122 in the appropriate location.

The address of the start of the attribute file is 22528 or 5800 hex. It can be represented by a grid of 32 column by 24 rows. The start of the file being in the top left-hand corner. If we wished to find the address of a given pair of co-ordinates (row 0, column 0 is in the top left-hand corner), then we could use the following piece of code:

```
        ;*****************************
        ; FIND ATTRIBUTE ADDRESS
        ; B CONTAINS THE ROW NUMBER
        ; C CONTAINS THE COLUMN NUMBER
        ; ON EXIT HL CONTAINS ADDRESS
CL-ATTR: LD     L,B
         LD     H,0
         ADD    HL,HL
         ADD    HL,HL
         ADD    HL,HL
         ADD    HL,HL
         ADD    HL,HL    ;FIND ROW TIMES 32
         LD     B,0
         ADD    HL,BC    ;ADD COLUMN OFFSET
         LD     BC,5800H ;START OF ATTRIBUTES
         ADD    HL,BC    ;ADD START OF ATTRIBUTES
         RET             ;ADDRESS NOW IN HL
```

As you can see we multiply the row number by 32 and by a series of ADD HL, HL instructions. Finally, we add the column offset. This isn't the quickest way of calculating the address but it is the easiest. If you read the chapter on shifting and rotating you way wish to calculate another way of finding the address using bit manipulation. This routine can be used for PEEKing or POKEing at the attribute file. If we wanted to look at the contents of a given row and column we could use the code:

```
PEEK:  CALL  CL-ATTR  ;CALCULATE ADDRESS AT
                      ;ROW B,COLUMN C
       LD    A,(HL)   ;PUT CONTENTS IN A REGISTER.
```

If we wanted to POKE red INK, green PAPER at column 22, row 5 we could write the code:

```
POKE: LD    A,22H     ;CODE FOR RED INK,GREEN PAPER
      LD    BC,0516H  ;SET BC TO ROW 5,COLUMN 22
      CALL  CL-ATTR   ;FIND ADDRESS
      LD    (HL),A    ;POKE VALUE IN.
```

The Screen File

The screen file is a little more complicated than the attribute file! It consists of three sections of 8 character rows. Each character row is made up of 32 characters split into 8 pixel lines. Since we are working with eight bit bytes the resolution of the screen is 256 by 192. The resolution of a screen determines the number of little dots that make up the data on the screen. The higher the resolution, the more detailed the pictures on the screen. To determine an address on the screen for a given pair of co-ordinates is not an easy matter. If we look at the bit pattern for a screen address:

010sslllrrrccccc

ss is the section number, 0 being the top third, 1 being the middle and 2 the bottom third.3 indicates an address in the attribute file. lll is the pixel line number (0–7) within a character. rrr is the line number within a section (0–7) and ccccc is the column number (0–31).

Using this pattern we can determine an address anywhere on the screen, even down to one single bit. The section number is contained in the top two bits of the row. The pixel line number is also contained in the row, this time in the middle three bits. The pixel line number is the last three bits of the row. The column is represented by a number 0–255. The range 0 to 255 is used because the routine is designed to give the address and bit position of any pixel on the screen. The data for the routine is as follows:

```
row    ssrrrlll
col    cccccbbb
```

where all the letters have their previous meanings and bbb is the bit position of the pixel.

Now let's look at the routine to calculate the screen address for any given row and column. The B register is LOADed with a row

number in the region 0–191, where row 0 is at the top of the screen. The C register is LOADed with the column number in the range of 0–255. After executing the routine the HL pair will contain the address on the screen, and the A register will contain the bit position (0–7) within that address.

```
        ; FIND PIXEL ADDRESS
        ; B CONTAINS Y COORD
        ; C CONTAINS X COORD
        ; ON EXIT HL CONTAINS ADDRESS
        ; A CONTAINS PIXEL NUMBER
PIXADD:
        LD    A,B          ;GET Y REGISTER
        RRA
        SCF
        RRA
        RRA                ;X1x..  MOVE DOWN SECTION
        AND   58H          ;MASK OFF SECTION. 010SS
        LD    H,A          ;SAVE IN H REG
        LD    A,B          ;GET Y AGAIN.
        AND   7            ;WORK OUT PIXEL LINE WITHIN
                           ;CHARACTER
        ADD   A,H          ;ADD PREVIOUS RESULT
        LD    H,A          ;SAVE IN H REG
        LD    A,C
        RRCA
        RRCA
        RRCA
        AND   1FH
        LD    L,A          ;MOVE DOWN COLS 0-31
        LD    A,B          ;GET ROW NUMBER
        AND   38H          ;MASKING OFF ROW NUMBER
        ADD   A,A
        ADD   A,A
        OR    L
        LD    L,A
        LD    A,C          ;GET BIT NUMBER
        AND   7            ;A CONTAINS BIT NUMBER
        RET
```

This routine could be used to PLOT points on the screen since the PLOT command in BASIC is limited to accessing only 256 by 176 points. To do this we call the PIXADD routine then rotate the pixel to the bit

position we want. So our program to PLOT a point at the co-ordinates in the BC register pair would be:

```
        ; PLOT A PIXEL AT THE COORDINATES
        ; X,Y
        ; B=Y COORD AND C=X COORD
PLOT:
        CALL PIXADD     ;Find address for co-ords BC
        LD   B,A        ;Put bit position in the B
                        ;register
        INC  B          ;now in the range 1-8
        XOR  A          ;set A to zero and clear
                        ;carry flag.
PIX:
        RRA
        DJNZ PIX        ;move pixel dot to position
                        ;required
        XOR  (HL)       ;and place it at the address
        LD   (HL),A
        RET
```

As seen I have used an EXCLUSIVE OR to place the dot on the screen. This has the effect of turning a pixel on if there wasn't already a dot at the calculated address or turning the pixel off, if it was already set.

When we PRINT a character onto the screen within a character boundary the offset for each byte of data which makes up a character is 256 so it is a simple matter to have a loop such as:

```
        ;DE POINTS TO DATA WHICH WE WANT TO PRINT
        ;HL POINTS TO SCREEN ADDRESS.TOP OF CHARACTER
        ;BOUNDARY
        LD   B,8        ;8 BYTES OF DATA
NXTPL:  LD   A,(DE)     ;GET DATA
        LD   (HL),A     ;PLACE DATA ON SCREEN
        INC  DE         ;POINT TO NEXT DATA
        INC  H          ;POINT TO NEXT PIXEL LINE
        DJNZ NXTPL      ;DO NEXT PIXEL LINE
```

The INC H instruction is the same as adding 256 to the HL register pair and has the effect of getting the next pixel line address below. The offset is always 256 only if we are within a character boundary. If this is not true we have to use the following routine below which I have called INCY:

```
INCY:
        INC  H
        LD   A,H
        AND  7
        RET  NZ             ;WITHIN CHAR BOUNDARY
        LD   A,H
        SUB  8
        LD   H,A
        LD   A,L
        ADD  A,32           ;NEXT CHAR LINE DOWN
                            ;(WITHIN SECTION)
                            ;ADD 32 DECIMAL
        LD   L,A
        RET  NC             ;DEF WITHIN SECTION
                            ;NEXT SECTION DOWN
        LD   A,H
        ADD  A,8
        LD   H,A
        XOR  58H            ;01011000 BINARY
        RET NZ              ;IS THERE A WRAPAROUND
                            ;NEEDED?
        LD   H,40H
        RET
```

The routine could be written to run a little quicker. It seems a waste of time to first subtract eight from the H register if we have gone over a character boundary and then to add eight back. The reason we have done this is to incorporate a wrap around effect. This means anything which is printed over the bottom of the screen will appear on the top.

There is also a relationship between the address of the attribute file and the screen file. Study the bit patterns for the high byte of the start of each section on the screen file and the corresponding bit patterns on the attribute file. The table below shows how the high bytes of the addresses relate:

screen addr	screen bit pattern	attr addr	attr bit pattern
40H	01000000	58H	01011000
48H	01001000	59H	01011001
50H	01010000	5AH	01011010

To get the corresponding attribute address from a given screen address shift down the high byte to the left three times and then set bits three and four of the high byte.

```
;THIS PIECE OF CODE GIVES
;THE ADDRESS OF THE ATTRIBUTE FILE IN THE
;HL REGISTER PAIR
;FOR A GIVEN ADDRESS ON THE DISPLAY FILE
;IN THE HL REGISTER PAIR.
LD   A,H          ;010SS000
SRA  A            ;0010SS00
SRA  A            ;00010SS0
SRA  A            ;000010SS
OR   50H          ;010110SS
LD   H,A
RET
```

ANIMATION

Using the INCY routine we can write another routine which allows us to PRINT a character at any pixel position on the screen. Usually when we PRINT a character in BASIC the character is placed on the standard 32 by 24 grid. Therefore there are only 768 positions at which we can place that character. If we were to write a game using the BASIC PRINT statement movement of characters is limited to moving horizontally eight bits at a time and vertically eight pixel lines at a time. The following machine code routine will demonstrate how to move objects around the screen smoothly using pixel movement.

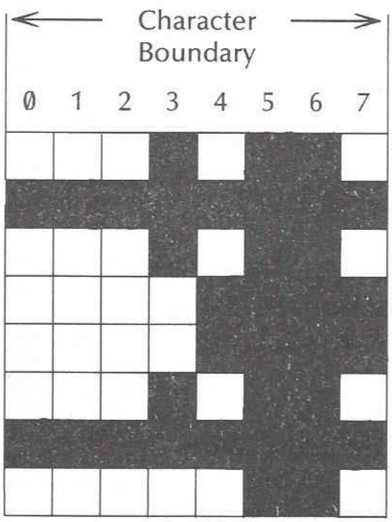
The routine in the ROM which deals with PRINTing characters in BASIC calculates the screen address for a given pair of co-ordinates. It is then a simple matter of lacing the eight bytes of data which make up a character onto the screen. The screen is constructed in such a way that each vertical line, where the character is to be placed, is 256 bytes below the last pixel line. However, this offset changes when we are PRINTing over a character or section boundary. If we wanted to draw a character on any of the 192 pixel lines we would need to keep using the INCY routine to find the addresses of successive pixel lines.

Therefore if our character stayed within a character boundary vertically then the following routine would print a character to the screen. The screen address is pointed to by the HL register pair and the character data is pointed to by the DE register pair.

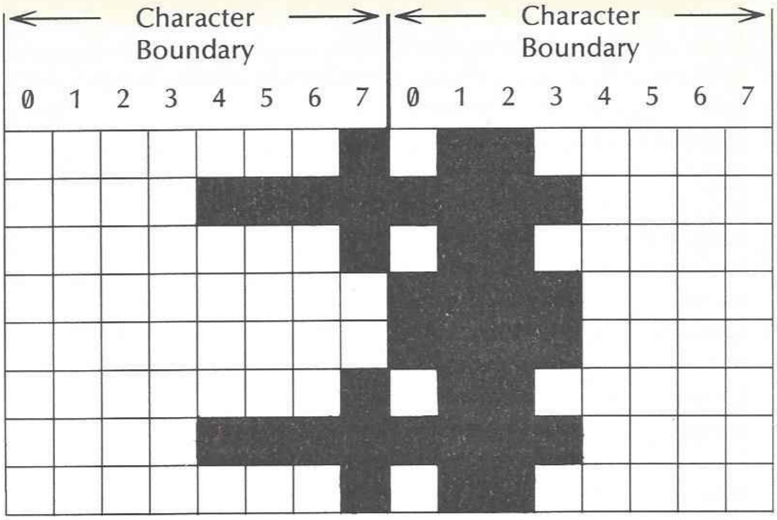
```
BOUNDH: LD     B,8        ;GET DATA COUNT
NXC:    LD     A,(DE)     ;GET CHARACTER DATA
        LD     (HL),A     ;AND PLACE ON THE SCREEN
        CALL   INCY       ;NEXT PIXEL LINE DOWN
        INC    DE         ;POINT DE TO NEXT
                          ;CHARACTER DATA
        DJNZ   NXC        ;DO 8 TIMES
```

As you can see this portion of code is similar to the first routine we used to PRINT a character to the screen. The exception is that the INC H instruction is replaced by calling the routine INCY which calculates the address of the next pixel line down.

The next problem we have to overcome when PRINTING a character on the screen is to deal with its horizontal position. When we want to print an eight bit character, at any of the 256 bits, we may sometimes overlap between two character boundaries. This means that if we can calculate the bit position where the object is to be placed within one of the 32 horizontal positions on the screen we can scroll the eight bit number which makes up one line of the character through two bytes which we then PRINT onto the screen. Look at the following two diagrams. Diagram A shows a space ship being PRINTED within a character boundary. The data only occupies one byte for each horizontal line. When we wish to PRINT an eight by eight bit object at any horizontal pixel position then we could get an overlap onto the adjacent character position as shown in diagram B. An overlap will occur seven in every eight horizontal bit positions. To find a character's bit position simply get it's X co-ordinate and mask off the bottom three bits by ANDing it with seven. Remember, that this 'bit position' is different from the one we use to describe bit instructions such as SET,RESET and BIT. This time the bit position starts from the left hand side of the byte.



8 by 8 pixel character within boundary



8 by 8 pixel character over boundary

If given the bit position in which a character lies, then to obtain its two byte equivalent, get the object data to be printed and scroll it from left to right within two bytes. This 'scrolling' from left to right of a 16 bit number is identical to dividing the number by 2. This was explained in the chapter on rotating and shifting. The following piece of code divides a two byte number in the A register (the high part) and the C register (the low part) by 2.

```
LD      C,0   ;clear low byte first
SRL     A     ;scroll A reg from left to right into carry
RR      C     ;scroll c register from left to right through
              ;carry
```

Notice how we clear the low byte of the number by LOADing the C register with 0. We would of course do this scrolling until we reach the bit position which we require. Therefore, if the B register contained the bit position we would find the two byte number by using the code:

```
; B REGISTER CONTAINS THE BIT POSTION
        LD      A,B             ;PLACE B REGISTER INTO
                                ;THE A REGISTER
        AND     A               ;TEST FOR BIT POSTION=0

        JR      Z,BOUND         ;WITHIN ONE CHARACTER
                                ;BOUNDARY SO DEAL WITH
                                ;THIS CASE AT LABEL BOUND
```

```
        LD      C,0             ;CLEAR RIGHT BYTE FIRST

GETB:   SRL     A               ;SCROLL A REGISTER
        RR      C               ;RIGHT THROUGH THE
                                ;C REGISTER
        DJNZ    GETB            ;UNTIL WE GET INTO
                                ;THE REQUIRED BIT
                                ;POSTION
```

Notice that before we scroll the character we test that the bit position is zero. If the bit position was zero then this means that our character is within a boundary so we deal with this at the lable BOUND. If we did not do this and carried on through to scroll the data then we would find that we would end up scrolling the data 256 times.

After we have our two new characters which make up the object then it is simply a case of placing them on the screen. If, for example, the HL register pair was pointing to the screen address where we wanted to place the character then we would place the data at HL and HL+1

```
;DRAW OBJECT ONTO THE SCREEN
;AT THE ADDRESS IN THE HL PAIR


        LD      (HL),A          ;PLACE LEFT HAND SIDE
                                ;OF THE OBJECT
        INC     HL              ;POINT TO NEXT CHARACTER
                                ;BOUNDARY
        LD      (HL),C          ;PLACE RIGHT HAND SIDE
                                ;OF THE OBJECT
```

To animate, simply DRAW the object onto the screen and to move it, remove the object from its previous position. Then update its new position and DRAW it to the screen. The following machine code routine DRAWS and animates nine space ships on the screen. Each one follows a movement pattern. The object can move in any one of four directions. Direction one indicates that the ship is moving right, two left, four down and eight up. The movement pattern DIRTAB is a table of directions which the ship follows and ends with 255 or FF hex. The ships start at different locations in the table so that the movements are not synchronised.

Each ship has three bytes of data starting from SHIPTB, to represent its X and Y co-ordinates and an offset position or vector count

pointing to a direction within the movement table. If a ship had a starting offset of 12 the first direction it would use is at the address DIRTAB+12. This contains a one and means that the ship would move right. When the ship comes to the end of the direction table (signified by the byte FF hex) it resets its vector count to zero thus pointing to the first byte of the direction table.

The IY register is used in this routine to point to each of the ships data. At the start of the program the ships are first drawn onto the screen. The main routine which deals with drawing characters uses the ROM routine PIXADD (22AA hex). When given the X and Y co-ordinates on the screen it will RETURN the screen address in the HL pair and the A register holds the bit position within a byte. The co-ordinates X=0, Y=0 start the A register at the left hand side of the screen 22 lines from the top. Therefore, we only have 176 pixel lines vertically to which to DRAW the objects. Of course if you wanted you could us my PIXADD routine which makes full use of the 256 by 192 screen.

Notice in the routine PRTCHR how instead of LOADing in the data character bytes directly I first exclusive OR the data with the contents of the screen. This is extremely useful for I can if I wish MOVE over 'background objects' on the screen without corrupting the data. It is like using the OVER 1 command in BASIC. As well as leaving the background it also serves a useful purpose for effacing the ship from the screen when MOVing it to a new position. Since we are using the XOR instruction this will turn off any bits that are all ready on and turn on bits already off.

Assember Listing

```
;                MOVEMENT ROUTINE
;
;
;
;
                                   ;EXAMPLE OF PIXEL MOVEMENT.

         ORG     28000D
         JP      START             ;START THE PROGRAM
```

```
SHIP:   DB      22,255,22,15,15,22,255,22
        ;DATA FOR SPACE SHIP


PRINOBJ:                        ;SUBROUTINE PRINT OBJECT
                                ;PRINTS AN OBJECT AT X,Y
                                ;B REG=Y VALUE
                                ;C REG=X VALUE


        PUSH    BC
        CALL    PRTCHR
        POP     BC
        RET


PRTCHR:                         ;PRINTS A SHIP ON ANY
                                ;PIXEL POSTION


        LD      IX,SHIP

        CALL    22AAH           ;FIND PIXEL ADDRESS

;ROUTINE AT 22AAH FINDS THE ADDRESS ON SCREEN FOR  A
;GIVEN X,Y CO-ORDS IN C REG AND B REG
;THE A REG IS RETURN WITH THE START OF PIXEL POSTION
;WITHIN THAT BYTE THIS ROUTINE IS ONLY LIMITED FOR Y
;BEING BETWEEN 0 AND 175 INSTEAD OF THE EXPECTED 0-191
;ALSO THE CO-ORD 0,0 START IN THE BOTTOM LEFT HAND SIDE.


        LD      E,A             ;SAVE BIT POSTION IN E
                                ;REGISTER
        LD      D,8             ;PIXEL LINE COUNT
        AND     A               ;TEST FOR ZERO PIXEL
                                ;POSTION
        JR      Z,WBOUND        ;WITHIN BOUNDARY SO JUMP
```

```
LINE:   LD      B,E             ;GET BIT POSTION IN
                                ;B REGISTER

;SCROLL DATA B TIMES INTO REGS A AND C
        LD      A,(IX+0)        ;GET DATA
        LD      C,0             ;CLEAR RIGHT HAND SIDE

SCROLL:
        SRL     A               ;SCROLL DATA DOWN TO BIT
                                ;POSTION
        RR      C
        DJNZ    SCROLL

        XOR     (HL)            ;MERGE LEFT HAND PART OF
                                ;DATA IN!
        LD      (HL),A

        INC     HL              ;TO THE RIGHT MARCH!
        LD      A,C             ;GET SECOND CHAR
ONE:    XOR     (HL)
        LD      (HL),A          ;MERGE SECOND CHAR IN.
        DEC     HL
        INC     IX              ;BUMP NEXT DATA
        CALL    INCY            ;FIND ADDRESS OF NEXT
                                ;PIXEL LINE
        DEC     D               ;HAVE WE DONE THE 8 BYTES?
        JR      NZ,LINE         ;NO! SO DO NEXT LINE
        RET                     ;YES THEN RETURN.

WBOUND: LD      B,D             ;LOAD B WITH 8
                                ;(PIXEL LINE COUNT)
XBOUND:
        LD      A,(IX+0)        ;GET DATA
        XOR     (HL)            ;MERGE IN WITH DATA
        LD      (HL),A          ;ALREADY ON THE SCREEN
        CALL    INCY            ;NEXT PIXEL LINE DOWN
        INC     IX              ;NEXT DATA BYTE
        DJNZ    XBOUND          ;REPEAT 8 TIMES
        RET
```

```
INCY:                              ;FINDS ADDRESS OF NEXT
                                   ;PIXEL LINE ON THE SCREEN
        INC     H
        LD      A,H
        AND     7
        RET     NZ                 ;WITHIN CHAR BOUNDARY
        LD      A,H
        SUB     8
        LD      H,A

        LD      A,L
        ADD     A,32D              ;NEXT CHAR LINE DOWN
                                   ;(WITHIN SECTION)
        LD      L,A
        RET     NC                 ;CHAR WITHIN SECTION

                                   ;NEXT SECTION DOWN


        LD      A,H
        ADD     A,8
        LD      H,A
        XOR     88                 ;01011000
        RET     NZ
        LD      H,40H              ;WRAP AROUND EFFECT
        RET




DIRTAB:
        DB      8,8,8,8,8,8,8,8,8,8,8,8
        DB      1,1,1,1,1,1,1,1,1,1,1,1
        DB      4,4,4,4,4,4,4,4,4,4,4,4
        DB      2,2,2,2,2,2,2,2,2,2,2,2
        DB      0FFH               ;END OF DIRECTION TABLE
```

```
GETDIR:
        LD      HL,DIRTAB       ;POINT TO DIRECTION TABLE
        LD      E,(IY+VECTCN)   ;GET SHIPS POINTER
        LD      D,0
        ADD     HL,DE           ;POINT TO DIRECTION
        LD      A,(HL)          ;GET DIRECTION
        CP      0FFH            ;IS THIS THE END OF
                                ;THE TABLE
        JR      NZ,MOVEIT       ;NO THEN MOVE SHIP!
        XOR     A               ;SET A TO ZERO
        LD      (IY+VECTCN),A   ;SET VECTOR COUNT TO ZERO
        RET


MOVEIT:
        INC     (IY+VECTCN)     ;INCREASE VECTOR COUNT
                                ;FOR NEXT GO
        CP      8               ;GOING UP
        JP      Z,UPD
        CP      4
        JP      Z,DOWND         ;GOING DOWN
        CP      1
        JP      Z,RIGHTD        ;GOING RIGHT
        CP      2
        JP      Z,LEFTD         ;GOING LEFT
        RET

UPDAA:
        CALL    PRINOBJ         ;REPRINT SHIP
                                ;AT NEW POSTION
        LD      (IY+XPOS),C     ;SAVE NEW XPOSTION
        LD      (IY+YPOS),B     ;SAVE NEW YPOSTION
        RET


LEFTD:
        CALL    PRINOBJ
        DEC     C
        JP      UPDAA

RIGHTD:
        CALL    PRINOBJ
        INC     C
        JP      UPDAA
```

```
UPD:
        CALL    PRINOBJ
        DEC     B
        JP      UPDAA

DOWND:
        CALL    PRINOBJ
        INC     B
        JP      UPDAA



XPOS    EQU     0               ;XPOS OFFSET
YPOS    EQU     1               ;YPOS OFFSET
VECTCN  EQU     2               ;VECTOR COUNT OFFSET

NUM     EQU     9               ;NUMBER OF SHIPS
LEN     EQU     3               ;LENGTH OF DATA FOR TABLE

;SHIP TABLE
;3 BYTES PER SHIP
;1ST BYTE =X CO-ORD
;2ND BYTE =Y CO-ORD
;3RD BYTE =VECTOR COUNT

SHIPTB:
        DB      100,100,8
        DB      120,80,7
        DB      55,45,14
        DB      30,30,20
        DB      40,30,1
        DB      130,130,24
        DB      140,140,20
        DB      140,118,2
        DB      140,150,2
```

```
START:
        DI                          ;DISABLE INT
        LD      B,NUM               ;LOAD B REGISTER WITH
                                    ;NUMBER OF SHIPS
        LD      IY,SHIPTB           ;IY POINTS TO START
                                    ;OF SHIP TABLE
DRAW:   PUSH    BC                  ;SAVE SHIP COUNTER
        LD      B,(IY+YPOS)         ;GET Y CO-ORD
        LD      C,(IY+XPOS)         ;GET X CO-ORD
        CALL    PRINOBJ             ;AND PRINT
        LD      DE,3                ;DE CONTAINS OFFSET
        ADD     IY,DE               ;POINT TO NEXT SHIP'S DATA
        POP     BC                  ;RESTORE COUNTER
        DJNZ    DRAW                ;DRAW NEXT SHIP

MOVE:
        LD      IY,SHIPTB           ;POINT TO SHIP TABLE
        LD      B,NUM               ;NUMBER OF SHIPS
NXT:
        PUSH    BC                  ;SAVE COUNTER
        LD      B,(IY+YPOS)         ;GET Y CO-ORD
        LD      C,(IY+XPOS)         ;GET X CO-ORD
        CALL    GETDIR              ;GET DIRECTION  AND MOVE
        LD      DE,3                ;PLACE OFFSET IN DE
        ADD     IY,DE               ;AND POINT TO NEXT SHIPM DATA
        POP     BC                  ;RESTORE SHIP COUNTER
        DJNZ    NXT                 ;MOVE NEXT SHIP
        JR      MOVE                ;FOREVER AND SO ON.......
        END
```

Hexadecimal Listing

```
6D60    C3  5D  6E  16  FF  16  0F  0F
6D68    16  FF  16  C5  CD  71  6D  C1
6D70    C9  DD  21  63  6D  CD  AA  22
6D78    5F  16  08  A7  28  1C  43  DD
6D80    7E  00  0E  00  CB  3F  CB  19
6D88    10  FA  AE  77  23  79  AE  77
6D90    2B  DD  23  CD  A8  6D  15  20
6D98    E5  C9  42  DD  7E  00  AE  77
```

6DA0	CD	A8	6D	DD	23	10	F4	C9
6DA8	24	7C	E6	07	C0	7C	D6	08
6DB0	67	7D	C6	20	6F	D0	7C	C6
6DB8	08	67	EE	58	C0	26	40	C9
6DC0	08	08	08	08	08	08	08	08
6DC8	08	08	08	08	01	01	01	01
6DD0	01	01	01	01	01	01	01	01
6DD8	04	04	04	04	04	04	04	04
6DE0	04	04	04	04	02	02	02	02
6DE8	02	02	02	02	02	02	02	02
6DF0	FF	21	C0	6D	FD	5E	02	16
6DF8	00	19	7E	FE	FF	20	05	AF
6E00	FD	77	02	C9	FD	34	02	FE
6E08	08	CA	34	6E	FE	04	CA	3B
6E10	6E	FE	01	CA	2D	6E	FE	02
6E18	CA	26	6E	C9	CD	6B	6D	FD
6E20	71	00	FD	70	01	C9	CD	6B
6E28	6D	0D	C3	1C	6E	CD	6B	6D
6E30	0C	C3	1C	6E	CD	6B	6D	05
6E38	C3	1C	6E	CD	6B	6D	04	C3
6E40	1C	6E	64	64	08	78	50	07
6E48	37	2D	0E	1E	1E	14	28	1E
6E50	01	82	82	18	8C	8C	14	8C
6E58	76	02	8C	96	02	F3	06	09
6E60	FD	21	42	6E	C5	FD	46	01
6E68	FD	4E	00	CD	6B	6D	11	03
6E70	00	FD	19	C1	10	EE	FD	21
6E78	42	6E	06	09	C5	FD	46	01
6E80	FD	4E	00	CD	F1	6D	11	03
6E88	00	FD	19	C1	10	EE	18	E6
6E90	6E	FE	01	CA	2D	6E	FE	02
6E98	CA	26	6E	C9	CD	6B	6D	FD
6EA0	71	00	FD	70	01	C9	CD	6B
6EA8	6D	0D	C3	1C	6E	CD	6B	6D
6EB0	0C	C3	1C	6E	CD	6B	6D	05
6EB8	C3	1C	6E	CD	6B	6D	04	C3
6EC0	1C	6E	64	64	08	78	50	07
6EC8	37	2D	0E	1E	1E	14	28	1E
6ED0	01	82	82	18	8C	8C	14	8C
6ED8	76	02	8C	96	02	F3	06	09

6EE0	FD	21	42	6E	C5	FD	46	01
6EE8	FD	4E	00	CD	6B	6D	11	03
6EF0	00	FD	19	C1	10	EE	FD	21
6EF8	42	6E	06	09	C5	FD	46	01

There are many improvements that could be made to this routine. You could easily add other directions, other direction tables and other characters. In fact if you want to be more ambitious you could make the objects of variable width and height. In addition improve the method of placing data on the screen by DRAWing from the bottom upwards instead of from the top downwards. This method of DRAWing objects will reduce flicker from the raster by catching the object as it is being DRAWN. All you have to work out is a routine similar to INCY but find the next pixel line above for a given screen address. I do know of a couple of ways to achieve this but I am not going to spoil your fun by explaining it to you!

11 Interrupts on the Spectrum

Have you ever wished that your computer could execute more than one program at once? Well, this chapter will explain how, in effect, you can double the power of your Spectrum by seemingly RUNning two programs at once!

Interrupts on the Z80 chip serve similar purposes to those on other processors. They tell the computer that an external device, such as a disk drive, printer, keyboard or modem requires some attention. Take, as an example, the case where we have linked up a printer printing out data to our computer.

There are two ways of checking whether the printer is ready to get a character from the microprocessor. The inefficient way is to use a loop which has a description like this:

```
WAIT:
        IS PRINTER READY?
        ANSWER=NO THEN GO TO WAIT

        ANSWER=YES THEN GET NEXT CHARACTER: SEND IT
                  TO THE PRINTER: GO TO WAIT
```

As you can see the above method 'polls' the printer continually to see if it is ready for the next character. Most of its time is spent in this loop waiting for the printer, so a lot of CPU time is wasted! Wouldn't it be fine if we could continue with other parts of the program and only send characters when the printer is ready? Well we can by using interrupts! Your Spectrum uses interrupts to get characters from the keyboard and update the frames system variable.

What is happening on a Spectrum is that your computer is running Spectrum BASIC. Frequently, (1/50 of a second to be precise or 1/60 of a second in N. America) it remembers where it is and what line it is running. It also recalls what address it is executing in the ROM or RAM and executes a routine in ROM which scans the keyboard. After it has done this it will go back to the address it was executing prior to interruption.

On the Z80 processor there are four kinds of interrupts. These interrupts are split into two categories called non-maskable and maskable interrupts. We shall be looking at just two of the maskable interrupts called mode 1 and mode 2 interrupts. (maskable means we can switch the interrupts off if we wish.)

mode 1 interrupts
Every time an interrupt occurs the processor pushes the current program counter onto the stack and jumps to location 0038 hex.

To exit out of this interrupt we must use a 'RET' (return) or 'RETI' (return from interrupt) instruction.

This mode of interrupt is actually the one used by the Spectrum during the scan for a key routine as described above.

mode 2 interrupts
This is the most powerful of the interrupts on the Z80 processor and is sometimes known as vectored processing.

In a mode 2 interrupt the programmer can specify up to 128 interrupts for other external devices.

This mode of interrupt revolves round a table which can contain up to 128 addresses. We can also have more than one table to deal with other external devices.

The start of a table is always on a page boundary of a 256 byte section of memory, i.e. 000H, 100H, 200H, C200H, etc. To tell the processor where the vector table is we LOAD the I (Interrupt) register with the high byte of the page number. For example if our vector table was at location C000 hex then we would tell the processor by executing:

```
DI          ;DISABLE INTERRUPTS
IM   2      ;SET UP INTERRUPT MODE 2
LD   I,C0H  ;LOAD I REGISTER WITH C0 HEX
EI          ;ENABLE INTERRUPTS
```

Note that we only need to specify the high byte as we are dealing with page boundaries. The second line of code tells the processor that we want to use mode 2 interrupts. The last instruction turns on the scanning of interrupts. If we wished to ignore any maskable interrupts at any time we would use the instruction:

```
DI          ;DISABLE INTERRUPTS
```

But wait! There is no instruction which allows us to LOAD the I register directly with a number. We can only LOAD the I register with the A register. We overcome this problem by using:

```
        LD   A,C0H  ;LOAD THE A REGISTER WITH C0H
        LD   I,A    ;AND PUT IT INTO THE I REGISTER.
```

In interrupt mode 2, when a device causes an interrupt, it provides an offset data number which is the low byte of the table. The offset points to a two byte address within the table to which the processor jumps (after first, stacking its current Program Counter).

```
TABLE  EQU      C000H

       IM       2
       LD       A,C0H
       LD       I,A         ;I REGISTER IS LOADED WITH HIGH
                            BYTE OF TABLE
       EI
       .
       .
       .

TABLE: DEFW     KEYBROU  ;ADDRESS OF KEYBOARD ROUTINE
       DEFW              ;ADDRESS OF PRINTER ROUTINE
       PRINROU
       .
       .
       .
           PUT ANY OTHER VECTORS HERE FOR OTHER DEVICES

KEYBROU:
           .
           .
           .
           RET
PRINROU:
           .
           .
           .
           RET
```

Now if an interrupt occurs and the data supplied is 0 then the processor pushes the current program counter on the stack and jumps to the address at C000H. If the data supplied was 02 then it would jump to the address at C002H which is the printer routine.

What if the data supplied was 01? If that happens a crash is likely to occur! Do you know why? The programmer has to program the

device to RETURN a valid data vector with its lowest bit set to zero i.e. always even!

Now what device can we program to cause an interrupt on a Spectrum? How do we program its eight bit vector number? The answer is we don't have too!! The Spectrum is not in conventional interrupt programming. Every 1/50 of a second an interrupt is generated by the ULA, one of the chips inside the computer. And at the time of the interrupt the data 0ff hex is passed to the microprocessor. If in interrupt mode 2 this will cause a jump to the address of the vector table currently pointed at by the I register plus 256 bytes.

Example:

```
        ORG  0C000H

INTINT: LD   A,0C0H          ;SET TABLE AT PAGE 0C0H
        LD   I,A             ;AND PLACE IN THE I REG
        IM   2               ;SET UP FOR INTERRUPT
                             ;MODE 2
        EI                   ;AND ENABLE
        RET


        ORG  0C0FFH          ;PLACE VECTOR AT PAGE+0FFH
        DEFW INTROU          ;ADDRESS OF
                             ;INTERRUPT ROUTINE


INTROU:
        DI                   ;STOP ANY MORE INTERRUPTS
        PUSH HL              ;SAVE HL
        LD   HL,5B00H        ;POINT TO THE ATTRIBUTE
                             ;FILE
        LD   (HL),255        ;AND SHOW SOME COLOUR
        POP  HL              ;RESTORE HL
        JP   0038H           ;JUMP BACK TO BASIC.
```

In our first example we notice that in our interrupt routine we disable the interrupt. This is only necessary when an interrupt is longer than 1/50 of a second so that we don't interrupt an interrupt! The JP 0038H jump to 0038 hex is the jump to BASIC keyboard scan routine. This scans the keyboard, updates the frame count and then

enables the interrupts. If we didn't wish to RETURN to BASIC then we would end an interrupt routine with:

```
      EI     ;ENABLE INTERRUPT
      RET    ;RETURN FROM INT
OR    EI
      RETI
```

Due to a hardware quirk in the Spectrum the value of the I register is limited to certain values 0–16 and 32–64. This means that for the 16K models we can only have our vector table in the page address 0–16 which is ROM!

There is, however, an end of page value which has a two byte value jumping out to RAM. This is page 28 hex.

LINKING THE INTERRUPT WITH THE RASTER

Re-set your Spectrum and type in the following program:

```
        ORG     30000

RASTER: HALT
        LD      A,1
        OUT     (0FEH),A
        LD      HL,500H         ;**EXPERIMENT**
                                ;WITH THIS VALUE!

LOOP:   DEC     HL
        LD      A,L
        OR      H
        JR      NZ,LOOP

        LD      A,2
        OUT     (0FEH),A
        JR      RASTER

        END
```

Here's a hexadecimal listing of the same program:

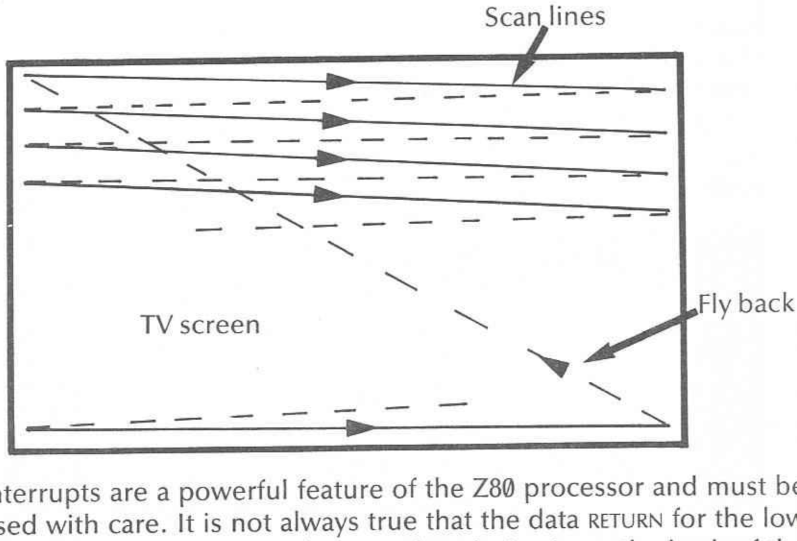
```
7D00  76 3E 01 D3 FE 21 00 05
7D08  2B 7D B4 20 FB 3E 02 D3
7D10  FE 18 ED 00
```

When you run the program you should get a BORDER split into two colours blue and red

The HALT instruction on the Z80 is used to wait for an interrupt. The computer will wait at a HALT instruction until some external device causes an interrupt. In the case of the Spectrum the ULA causes the interrupt. Therefore, the effect of the HALT instruction is to wait 1/50th of a second. Of course if we disabled all interrupts by using the instruction DI then the computer would wait for ever unless a Non-maskable interrupt (one that can not be disabled) was activated. In our program we use the HALT instruction to link with the raster beam to cause a split in the BORDER colour. Objects can be drawn when the raster is at the top or flying back thus reducing screen flicker. A lot of game programmers use this technique when writing fast arcade games.

Try pressing the keys when RUNning this program. Notice how the BORDERS go up and down. Do you know why? It is due to the keyboard routine (which is called by the interrupt routine) taking different lengths of time to execute depending on which keys it finds pressed.

Every 1/50th of a second the computer reDRAWs the screen. The screen is updated by an election beam which scans across the pixels turning them on or off if they are set or re-set. The beam starts from the top left hand side of the screen and scans left to right across each line. After reaching the bottom the beam (or raster) flys diagonally back to the top left where it starts to update the screen again.



Interrupts are a powerful feature of the Z80 processor and must be used with care. It is not always true that the data RETURN for the low byte of the vector is 0FFH if some other device is on the back of the

Spectrum. When an interrupt occurs with a Kempston joystick on the Spectrum the data on the databus is the actual data returned from READing the joystick. In order to overcome this problem we could fill our vector table with an address in which the low and high bytes are the same: 8080h 7777h 1616h.

Remember that on the interrupt the PC register will jump to the location in the table. We could put our interrupt routine at that address or have another jump instruction. Unfortunately, because of hardware problems limiting the value of the I register, this method is impossible on the 16K model Spectrums.

We'll now give you listings for two routines that use interrupts. The first is a TRACE program that can be used to help you debug BASIC programs. Every 1/50th of a second an interrupt occurs causing the transfer of the program counter to address 38 hex. This is where the keyboard is scanned and other 'housekeeping' tasks performed. We can cause the interrupt transfer to point to a routine which looks at the system variables PPC and SUBPPC. These contain the line number and statement number which BASIC is currently executing. PRINTing the values of these variables to the screen tells the BASIC programmer which statements the program is interpreting and the sequence of execution. This provides an extremely valuable aid for mapping the flow of the program, which in turn can greatly assist debugging.

This is not like the true TRACE functions found on some other computers since we can only see what line we are executing every 1/50th of a second. Some of the faster BASIC statements could be missed. The trace function is enabled by typing the instruction RAND USR 32330 and disabled by typing RAND USR 32338. When you are running BASIC and the trace function is enabled then the line you are executing at the time of the interrupt is displayed on the top left hand side of the screen. To slow BASIC down as it is executing with the TRACE function press the 'Q' key. This is most useful as it sometimes gets difficult to see the line numbers being PRINTed.

Assember Listing

```
         ORG     32330D


         ;TRACE ROUTINE
         ;FOR 16K SPECTRUM


ZEROADD EQU     15360            ;ADDRESS FOR NUMERIC DATA
PCC     EQU     23621            ;LINE NUMBER EXECUTED
```

```
SUBPCC  EQU     23623           ;STATEMENT WITHIN LINE

TRON:
        LD      A,28H           ;LOAD A WITH 28H
        LD      I,A             ;AND PLACE IN THE
                                ;I REGISTER
        IM      2               ;SET UP INTERRUPT MODE 2
        EI                      ;AND ENABLE
        RET

TROFF:  IM      1               ;INTERRUPT MODE 1
        RET


        ORG     7E5CH           ;INTERRUPT ROUTINE
                                ;STARTS HERE


TRACE:  DI
        PUSH    AF              ;SAVE REGS
        PUSH    BC
        PUSH    DE
        PUSH    HL
        PUSH    IX

        LD      HL,(PCC)        ;LOAD PROGRAM POINTER
        LD      A,H
        INC     A
        JR      Z,SKIP

        LD      DE,16384
        CALL    CONV            ;PRINT NUMBER ON SCREEN.
        INC     DE

        LD      A,(SUBPCC)      ;GET SUB-LINE NUMBER.
                                ;ONE BYTE NUMBER
        LD      H,0             ;AND TRANSFER TO
                                ;HL REGISTER PAIR.
        LD      L,A
        CALL    CONV2
```

```
        LD      A,0FBH

        IN      A,(0FEH)
        RRA
        JR      C,SKIP

        CALL    WAIT
        CALL    WAIT
SKIP:
        POP     IX
        POP     HL
        POP     DE
        POP     BC
        POP     AF
        JP      0038H


WAIT:   LD      HL,00
        LD      DE,00
        LD      BC,00
        LDIR
        RET

PRDIGIT:                        ;HL POINTS TO SCREEN
                                ;ADDRESS
                                ;A CONTAINS DIGIT
                                ;NUMBER 0-9

        PUSH    DE              ;SAVE REGISTERS
        PUSH    BC
        PUSH    IX
        PUSH    HL


        LD      H,0             ;PUT CHARACTER OFFSET
        LD      L,A             ;IN HL REGISTER

        ADD     HL,HL           ;MULTIPLY BY 8
        ADD     HL,HL
        ADD     HL,HL
```

```
        EX      DE,HL
        LD      IX,ZEROADD      ;POINT TO START OF
                                ;NUMERIC DATA
        ADD     IX,DE           ;ADD OFFSET TO START OF
                                ;NUMBER
        EX      DE,HL
        LD      B,8             ;SET COUNTER

NXDAT:  LD      A,(IX)          ;GET NUMERICAL DATA
        LD      (DE),A          ;PLACE ON SCREEN
        INC     D               ;ADJUST SCREEN ADDRESS
        INC     IX              ;ADJUST DATA POINTER.
        DJNZ    NXDAT           ;DO NEXT DATA

        POP     HL
        POP     IX
        POP     BC              ;RESTORE REGISTERS
        POP     DE
        RET


DECT2:
        DEFW    1000
DECT3:  DEFW    100
        DEFW    10
        DEFW    1
CONV2:
        LD      IX,DECT3
        JR      NDIG2
CONV:   LD      IX,DECT2
NDIG2:  LD      B,(IX+1)
        LD      C,(IX+0)
        LD      A,'0'-1
        AND     A
CAR:    INC     A
        SBC     HL,BC
        JR      NC,CAR
        ADD     HL,BC
        CALL    PRDIGIT
        INC     DE
        INC     IX
```

```
        INC     IX
        DEC     C
        JR      NZ,NDIG2
        RET

        END
```

Hexadecimal Listing

```
7E4A   3E  28  ED  47  ED  5E  FB  C9
7E52   ED  56  C9  00  00  00  00  00
7E5A   00  00  F3  F5  C5  D5  E5  DD
7E62   E5  2A  45  5C  7C  3C  28  1D
7E6A   11  00  40  CD  CE  7E  13  3A
7E72   47  5C  26  00  6F  CD  C8  7E
7E7A   3E  FB  DB  FE  1F  38  06  CD
7E82   90  7E  CD  90  7E  DD  E1  E1

7E8A   D1  C1  F1  C3  38  00  21  00
7E92   00  11  00  00  01  00  00  ED
7E9A   B0  C9  D5  C5  DD  E5  E5  26
7EA2   00  6F  29  29  29  EB  DD  21
7EAA   00  3C  DD  19  EB  06  08  DD
7EB2   7E  00  12  14  DD  23  10  F7
7EBA   E1  DD  E1  C1  D1  C9  E8  03
7EC2   64  00  0A  00  01  00  DD  21

7ECA   C2  7E  18  04  DD  21  C0  7E
7ED2   DD  46  01  DD  4E  00  3E  2F
7EDA   A7  3C  ED  42  30  FB  09  CD
7EE2   9C  7E  13  DD  23  DD  23  0D
7EEA   20  E6  C9  CD  CE  7E  13  3A
7EF2   47  5C  26  00  6F  CD  C8  7E
7EFA   3E  FB  DB  FE  1F  38
```

The second of our two interrupt driven routines allows us to have an on-screen clock constantly telling us the time, even when we are RUNning a BASIC program. After placing the machine code routine in memory key in the BASIC listing below. This serves to set the time on the clock. After RUNning, the BASIC program clock should be constantly updated on the top right hand side of the screen. You can stop the clock at any time by entering NEW. This disables the clock by re-setting the interrupt mode to 1, thereby causing the Z80 to

branch off to 38 hex on every interrupt. To start the clock off again simply type RANDOMIZE USR 32330

BASIC Listing

```
  10 CLEAR 32325:LET T=32438
  20 INPUT "HOURS";H:LET H=INT(H
):IF H<0 OR H>12 THEN GO TO 20
  30 INPUT "MINS ";M:LET M=INT(M
):IF M<0 OR M>59 THEN GO TO 30
  40 IF H>9 THEN LET H=H+6
  50 IF M>9 THEN LET M=M+6*INT(M
/10)
  60 POKE T,H:POKE T+1,M:POKE T+
2,0:RANDOMIZE USR 32330
```

Assembler Listing

```
        ORG     32330D

        ;CLOCK ROUNTINE
        ;FOR 16K SPECTRUM

TRON:
        LD      A,28H           ;SET UP I REGISTER
        LD      I,A             ;TO PAGE 28 HEX
        IM      2               ;SET INTERRUPT MODE 2
        EI                      ;AND ENABLE
        RET

FRAMES:
        DB      0

        ORG     7E5CH           ;START OF INTERRUPT ROUTINE
CLOCK:
        DI                      ;DISABLE INTERRUPTS
        PUSH    AF              ;SAVE REGISTERS ON
        PUSH    BC              ;THE STACK.
        PUSH    DE
        PUSH    HL
        PUSH    IX

        LD      A,(FRAMES)      ;UPDATE 1/50 SECOND
                                ;COUNTER.
```

```
        INC     A
        LD      (FRAMES),A
        CP      50              ;HAVE WE COUNTED THROUGH
                                ;1 SEC?
        JR      NZ,PRCLOCK      ;NO,SO PRINT TIME ANYWAY.

        XOR     A               ;SET FRAMES
        LD      (FRAMES),A      ;TO ZERO.

        LD      DE,TIMLIM       ;GET BCD TIME LIMITS
        LD      HL,SECS         ;POINT TO TIMER COUNTERS
        LD      B,3             ;NUMBER OF COUNTERS TO
                                ;UPDATE.

NXBCD:  LD      A,(HL)          ;GET TIME COUNTER
        ADD     A,1             ;INCREASE BY ONE
        DAA                     ;BCD
        LD      (HL),A
        LD      A,(DE)          ;GET LIMIT
        CP      (HL)            ;HAVE WE REACHED LIMIT FOR
                                ;THAT TIMER?
        JR      NZ,PRCLOCK
;NO SO GO AND PRINT TIME.
        LD      (HL),0          ;RESET TIME COUNTER
        INC     DE              ;POINT TO NEXT TIME LIMIT
        DEC     HL              ;POINT TO NEXT TIME COUNTER
        DJNZ    NXBCD           ;DO NEXT DIGIT.

;AT THIS POINT THE HOURS ARE RESET TO ZERO.

        INC     HL              ;POINT TO HOURS DIGIT
        INC     (HL)            ;SET TO 1 O'CLOCK.

PRCLOCK:
        ;ROUTINE TO PRINT THE CLOCK ON THE SCREEN

        LD      HL,16384+31-8   ;TOP RIGHT HAND CORNER
        LD      DE,HRS          ;DE POINTS TO BCD DIGITS
        LD      B,3             ;COUNTER
```

```
NXT:    LD      A,(DE)          ;GET DIGIT
        LD      C,A             ;SAVE IN C REG
        AND     0F0H            ;GET FIRST DIGIT
        RRCA                    ;MOVE DOWN TO BITS 0-3
        RRCA
        RRCA
        RRCA
        CALL    PRDIGIT         ;PRINT DIGIT.
        INC     HL              ;POINT TO NEXT PART
                                ;OF SCREEN
        LD      A,C             ;GET DIGIT
        AND     0FH             ;MASK OFF BOTTOM 4 BITS.
        CALL    PRDIGIT         ;PRINT DIGIT
        INC     DE              ;POINT TO NEXT
                                ;TWO BCD DIGITS.
        INC     HL              ;ONE SPACE BETWEEN DIGITS.
        INC     HL
        DJNZ    NXT             ;DO NEXT DIGITS

        POP     IX              ;RESTORE REGISTERS
        POP     HL
        POP     DE
        POP     BC
        POP     AF
        JP      0038H


TIMLIM: DB      60H,60H,13H
HRS:    DB      0
MINS:   DB      0
SECS:   DB      0

ZEROADD         EQU     15744   ;ADDRESS OF START OF
                                ;NUMERIC DATA

PRDIGIT:
                                ;HL POINTS TO SCREEN
                                ;ADDRESS
                                ;A CONTAINS DIGIT
                                ;NUMBER 0-9

        PUSH    HL              ;SAVE REGISTERS
        PUSH    BC
```

```
        SLA     A                   ;MULIPLY DIGIT BY 8
        SLA     A
        SLA     A
        LD      B,0                 ;GET OFFSET
        LD      C,A                 ;IN BC REGISTER
        LD      IX,ZEROADD          ;POINT TO START OF
                                    ;NUMERIC DATA
        ADD     IX,BC               ;ADD OFFSET TO START OF
                                    ;NUMBER

        LD      B,8                 ;SET COUNTER

NXDAT:  LD      A,(IX)              ;GET NUMERICAL DATA
        LD      (HL),A              ;PLACE ON SCREEN
        INC     H                   ;ADJUST SCREEN ADDRESS
        INC     IX                  ;ADJUST DATA POINTER.
        DJNZ    NXDAT               ;DO NEXT DATA

        POP     BC                  ;RESTORE REGISTERS
        POP     HL
        RET

        END
```

Hexadecimal Listing

```
7E4A   3E  28  ED  47  ED  5E  FB  C9
7E52   00  00  00  00  00  00  00  00
7E5A   00  00  F3  F5  C5  D5  E5  DD
7E62   E5  3A  52  7E  3C  32  52  7E
7E6A   FE  32  20  1D  AF  32  52  7E
7E72   11  B3  7E  21  B8  7E  06  03
7E7A   7E  C6  01  27  77  1A  BE  20
7E82   08  36  00  13  2B  10  F1  23

7E8A   34  21  17  40  11  B6  7E  06
7E92   03  1A  4F  E6  F0  0F  0F  0F
7E9A   0F  CD  B9  7E  23  79  E6  0F
7EA2   CD  B9  7E  13  23  23  10  E9
7EAA   DD  E1  E1  D1  C1  F1  C3  38
7EB2   00  60  60  13  00  00  00  E5
7EBA   C5  CB  27  CB  27  CB  27  06
7EC2   00  4F  DD  21  80  3D  DD  09
```

```
7ECA   06  08  DD  7E  00  77  24  DD
7ED2   23  10  F7  C1  E1  C9  00  00
7EDA   00  00  F3  F5  C5  D5  E5  DD
7EE2   E5  3A  52  7E  3C  32  52  7E
7EEA   FE  32  20  1D  AF  32  52  7E
7EF2   11  B3  7E  21  B8  7E  06  03
7EFA   7E  C6  01  27  77  1A
```

12 Machine code miscellany

In this final chapter, I present a complete machine code game and a variety of routines. You can use these to enhance your BASIC programs or incorporate into your own machine code programs. The techniques we've seen in the course of the book are all represented here. Study of the programs should help you in writing your own and will increase your repertoire of routines and your program library.

BRICKOUT

This version of a venerable arcade game is one of my favourite programs. There is no better way to become proficient in machine code than by writing games. The object of this game is to knock three layers of 32 bricks away from the top part of the screen. The player controls a bat and directs a ball moving along the screen to knock down the coloured bricks. If the ball passes the bat then the player loses a life. The game ends when the player loses all his lives. The game listing has been broken into sections and heavily annotated to help you see the structure of the program.

The bat is controlled by the <Z> and <X> keys which make it move left or right respectively. The <CAPS SHIFT> key can be used to make the bat go twice as fast, (a 'Cheat Key' if you like!). The game program comes in two parts. One is in BASIC and the other in machine code. To start the game after entering the code you RUN the BASIC program which calls the machine code routine. When the player has lost the score is printed on the RETURN to BASIC as the variable SC and the user is asked if he wants another game. Here's the BASIC program:

```
  10 PRINT #0;"press a key when
ready"
  20 IF INKEY$="" THEN GO TO 20
  30 CLS
  40 RANDOMIZE USR 30000
  45 PLOT 0,175: DRAW 255,0: PLO
T 0,0: DRAW 0,175
  50 GO TO 10
```

The machine code routine comes in three main sections. The first initializes the score, the number of bricks left, the number of balls left and draws the screen. The second routine, MOVBAT, moves the user's bat, controlled by the <Z>, <X> and <CAPS SHIFT> keys. The last routine, MUBALL, deals with moving the ball around the screen, knocking out bricks, rebounding off the bat and walls, and updating the score. We'll break down the assembler listing, and give the whole hexadecimal listing at the end.

To start off the game we jump into the portion of code labelled START. This follows the initialisation:

```
        ORG     32000
        JP      START
BATYX:  DEFW    160FH           ;BATS  POSITION
BALLX:  DB      10H             ;BALLS X POSITION
BALLY:  DB      01H             ;BALLS Y POSITION
TPBLYX:
        DEFW    0               ;TEMP AREA
XINC:   DB      1               ;X MOVEMENT
YINC:   DB      1               ;Y MOVEMENT
LEVEL:  DB      4               ;LEVEL
                                ;NUMBER OF HALTS FOR DELAY
SCORE:  DEFW    0               ;SCORE
BALLS:  DB      0               ;NUMBER OF BALLS
HITS:   DB      0               ;NUMBER OF BRICKS HIT

PATTBL:

SPACE:  DB      0,0,0,0,0,0,0,0

;DATA FOR BALL

BALLCH: DB      3CH,7EH,0FFH,0FFH,0FFH,0FFH,7EH,3CH
```

```
;DATA FOR BAT

BATCHS: DB      3FH,7FH,0FFH,0FFH,0FFH,0FFH,7FH,3FH
        DB      0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH
        DB      0FCH,0FEH,0FFH,0FFH,0FFH,0FFH,0FEH,0FCH

;DATA FOR BRICK

BRICK1: DB      0FFH,81H,81H,81H,81H,81H,81H,0FFH

START:
        LD      A,2
        CALL    1601H           ;OPEN CHANNEL TWO
        XOR     A               ;SET A REGISTER TO ZERO
        OUT     (0FEH),A        ;SET BORDER TO BLACK

        LD      HL,0            ;RE-SET SCORE TO 0
        LD      (SCORE),HL
        LD      A,5
        LD      (BALLS),A       ;SET NUMBER OF BALLS TO 5
        LD      A,4
        LD      (LEVEL),A
        LD      A,96            ;NUMBER OF BRICKS
        LD      (HITS),A        ;PLACE IN HITS
        CALL    SETUP           ;SET UP BRICKS ON SCREEN
        CALL    RNDBAL          ;PLACE THE BALL ON THE
                                ;SOMEWHERE ON THE SCREEN
        LD      HL,1610H        ;INTIALIZE BATS X,Y CO-ORDS
        LD      (BATYX),HL
        CALL    PRTBAT          ;AND PRINT THE BAT


BATAGN:
        LD      A,(BALLS)       ;LOOK AT THE NUMBER
                                ;OF BALLS LEFT
        AND     A               ;IS IT ZERO?
        JR      Z,GMOVER        ;IF IT IS GO TO
                                ;DEAL WITH END OF GAME
;WE ARE STILL PLAYING
        CALL    MOVBAT          ;MOVE BAT
        CALL    MUBALL          ;MOVE BALL
```

```
        EI
        HALT                        ;WAIT FOR 1/50 OF A SECOND
        DI
        JR      BATAGN              ;KEEP PLAYING

GMOVER:
        LD      BC,(SCORE)          ;PASS SCORE TO
                                    ;BC REGISTER PAIR
        EI                          ;ENABLE INTERRUPTS
        RET                         ;AND RETURN TO BASIC
```

The games ends when there are no balls left, which causes the routine to jump to the label GMOVER. This gets the current score and places it into the BC register pair to be passed back to BASIC.

MOVBAT is used to control the movement of the bat according to the keys <Z> or <X> being pressed. If the user presses the key <CAPS SHIFT> then the routine goes back to the label FIVE,90 to move the bat again.

A call is made to either RIGHTTB or LEFTTB to move the bat right or left. After this the routine RETURNS to move the ball.

```
MOVBAT:
                                    ;MOVE PLAYERS BAT

        CALL    PRTBAT              ;PRINT BAT

        LD      A,0FEH              ;SET UP TO SCAN BOTTOM
        IN      A,(0FEH)            ;LEFT HAND SIDE OF KEYBOARD
        AND     1FH                 ;MASK OFF LOWER FOUR BITS
        CP      1FH                 ;SEE IF ALL BITS ARE SET
        RET     Z                   ;NO KEY PRESSED SO RETURN

        CALL    CLRBAT              ;CLEAR BAT OFF SCREEN
FIVE90:
        BIT     1,A                 ;IF PRESSED 'Z'
        CALL    Z,LEFTB             ;THEN MOVE LEFT
        BIT     2,A                 ;IF PRESSED 'X'
        CALL    Z,RIGHTB            ;THEN MOVE RIGHT
        BIT     0,A                 ;HAVE WE PRESSED SHIFT
        JR      NZ,BATPRT           ;KEY?IF NOT JUST PRINT BAT
        SET     0,A
        JR      FIVE90              ;TURN OFF SHIFT KEY
                                    ;HAVE ONE MORE GO
```

```
BATPRT:
        CALL    PRTBAT          ;PRINT BAT ON SCREEN
        RET                     ;AND RETURN
```

When moving the ball left or right a check must be made to make sure that the bat does not go off the screen. The variable BATXY holds the X,Y co-ordinate of the left hand side of the bat. The bat is made up of three characters.

```
RIGHTB:
        PUSH    AF              ;GOING RIGHT, SAVE AF PAIR
        LD      HL,(BATYX)      ;GET X,Y CO-ORD OF BAT
                                ;IN HL PAIR
        LD      A,1DH           ;LOAD A REGISTER WITH 29
        CP      L               ;TEST TO SEE IF WE HAVE
                                ;HIT THE RIGHT SIDE
        JR      Z,REDGE         ;HIT, SO DON'T UPDATE
        INC     L               ;INCREASE X CO-ORD
        LD      (BATYX),HL      ;AND SAVE
REDGE:
        POP     AF              ;RESTORE KEY STATUS
        RET                     ;AND RETURN

LEFTB:
        PUSH    AF              ;GOING LEFT, SAVE KEY STATUS
        LD      HL,(BATYX)      ;GET X,Y CO-ORDS
        LD      A,L             ;TEST IF HIT LEFT HAND SIDE
        AND     A               ;IE IF EQUAL TO 0
        JR      Z,LEDGE         ;HIT SO DON'T UPDATE
        DEC     L               ;DECREASE ONE OFF X CO-ORD
        LD      (BATYX),HL      ;AND SAVE
LEDGE:
        POP     AF              ;RESTORE KEY STATUS
        RET                     ;AND RETURN
```

The routine CLRBAT is used to remove the bat from the screen. To do this we PRINT the character SPACE which consists of zeros. While the routine PRTBAT is used to PRINT the bat to the screen. Both these routines call the routine PRTCH which PRINTS the character held in the A register. In this routine a CALL is made to two ROM routines. The routine at 0E9E hex calculates the screen address for a given Y co-ordinate. The routine at the address 0E88 hex calculates the attribute in the DE register pair for a given screen address.

```
CLRBAT:
        PUSH    AF              ;SAVE AF REGISTER
        LD      HL,(BATYX)      ;GET X,Y CO-ORD
        LD      BC,338H         ;SET INK AND PAPER
                                ;WHITE PAPER BLACK INK
CLRIT:                          ;B REGISTER IS
                                ;LOADED WITH 3
        PUSH    BC              ;SAVE CHAR CODE
        PUSH    HL              ;SAVE X,Y CO-ORD AND
                                ;COUNTER
        XOR     A               ;SET A TO ZERO
        CALL    PRTCH           ;PRINT SPACE
        POP     HL              ;RESTORE X,Y
        INC     L               ;POINT TO NEXT CHAR OF BAT
        POP     BC              ;RESTORE X,Y CO-ORD
                                ;AND COUNTER
        DJNZ    CLRIT           ;RUB OFF 3 CHARACTERS
        POP     AF              ;RESTORE AF REGISTER
        RET

PRTBAT:
        LD      HL,(BATYX)      ;GET X,Y CO-ORD
        LD      BC,339H         ;SET B=3 AND COLOUR TO
                                ;WHITE PAPER AND RED INK
        LD      A,2             ;INTIALIZE A REG TO FIRST
                                ;CHARACTER OF BAT

NEXBAT:
        PUSH    BC              ;SAVE COLOUR AND COUNTER
        PUSH    HL              ;SAVE X,Y CO-ORD
        CALL    PRTCH           ;PRINT PART OF BAT
        INC     A               ;NEXT PART OF BAT
        POP     HL              ;RESTORE X,Y
        INC     L               ;NEXT X POSTION OF BAT
        POP     BC              ;RESTORE COUNTER AND COLOUR
        DJNZ    NEXBAT          ;DO 3 TIMES
        RET                     ;AND RETURN

PRINTCHAR:
        ;H=Y L=X  A=CHAR NUMBER  C=COLOUR
```

```
PRTCH:
        PUSH    AF
        PUSH    BC              ;SAVE CHARACTER
        PUSH    HL              ;SAVE COLOUR
                                ;SAVE X,Y CO-ORDS
        PUSH    BC
        PUSH    AF              ;SAVE COLOUR
        PUSH    HL              ;SAVE CHARACTER
                                ;SAVE X,Y CO-ORDS
        LD      A,H
        CALL    0E9EH           ;LOAD A WITH Y CO-ORD
        POP     DE              ;CALCULATE SCREEN ADDRESS
        LD      D,0             ;PLACE X CO-ORD IN E REG
        ADD     HL,DE           ;PLACE 0 IN D
        EX      DE,HL           ;FIND SCREEN ADDRESS
                                ;AND PLACE IN DE
        POP     AF              ;GET CHARACTER CODE
        LD      BC,PATTBL       ;BC POINTS TO CHARACTER SET
        LD      H,0             ;LOAD H WITH 0
        LD      L,A             ;LOAD A WITH CHARACTER
                                ;NUMBER
        ADD     HL,HL           ;TIMES BY 2
        ADD     HL,HL           ;TIMES BY 4
        ADD     HL,HL           ;TIMES BY 8
        ADD     HL,BC           ;ADD CHARACTER
                                ;TABLE ADDRESS
                                ;HL NOW POINTS
                                ;TO CHARACTER DATA
        LD      B,8             ;LOAD B WITH DATA COUNT

NXTROW: LD      A,(HL)          ;GET CHARACTER DATA
        LD      (DE),A          ;AND PLACE ON SCREEN
        INC     HL              ;POINT TO NEXT CHARACTER
                                ;DATA
        INC     D               ;POINT TO NEXT PIXEL
                                ;LINE ON THE SCREEN
        DJNZ    NXTROW          ;DO THIS UNTILL
                                ;WE HAVE FINISHED
                                ;PRINTING THE CHARACTER
        EX      DE,HL           ;LET HL NOW BE
                                ;THE SCREEN ADDRESS
        CALL    0E88H           ;CALCULATE THE
                                ;ATTRIBUTE ADDRESS
        POP     BC              ;RESTORE COLOUR CODE
```

```
        LD      A,C             ;PLACE IN A REGISTER
        LD      (DE),A          ;SET ATTRIBUTE
        POP     HL              ;RESTORE X,Y CO-ORD
        POP     BC              ;RESTORE COLOUR CODE
        POP     AF              ;RESTORE CHARACTER
        RET                     ;RETURN FROM PRINTING
```

The routine SETUP is called only once: at the start of each new game. It is used to draw the bricks on the screen.

```
;SET START SCREEN

SETUP:
        CALL    0D6BH           ;CLEAR SCREEN
        LD      BC,2020H        ;32 GREEN BRICKS
        LD      A,5             ;PLACE BRICK CHAR IN A REG
        LD      HL,300H         ;START X,Y CO-ORD OF BRICKS
        CALL    NXCOL           ;DRAW BRICKS

        LD      BC,2018H        ;COLOUR =18H MAGENTA
        LD      HL,400H         ;Y=4   4 X=0
        CALL    NXCOL           ;DRAW BRICKS

        LD      BC,2030H        ;COLOUR =30H YELLOW
        LD      HL,500H         ;Y=5,X=0

NXCOL:
        CALL    PRTCH           ;PRINT BRICK
        INC     L               ;POINT TO NEXT X CO-ORD
        DJNZ    NXCOL           ;REPEAT 32 TIMES

        RET                     ;RETURN
```

PEEK is the routine which is used to detect any collision between the ball and any bricks or the bat. The X and Y co-ordinates are placed in the HL pair and after CALLing this routine the attribute or colour code is RETURNed in the A register.

```
PEEK:
                                ;RETURNS ATTRIBUTE
                                ;OF GIVEN X,Y (IN HL PAIR)
                                ;IN A REGISTER
        LD      A,L             ;PLACE X CO-ORD
                                ;IN A REGISTER
        LD      L,H             ;PLACE Y CO-ORD
                                ;IN L REGISTER
        LD      H,0             ;32 BIT NUMBER SO
                                ;PLACE 0 IN H
        ADD     HL,HL           ;TIMES BY 2
        ADD     HL,HL           ;TIMES BY 4
        ADD     HL,HL           ;TIMES BY 8
        ADD     HL,HL           ;TIMES BY 16
        ADD     HL,HL           ;TIMES BY 32
        LD      B,0             ;LOAD B REG WITH 0
        LD      C,A             ;PLACE X CO-ORD IN C REGISTER
        ADD     HL,BC           ;FIND OFFSET
        LD      BC,5800H
        ADD     HL,BC           ;CALCULATE ATTRIBUTE
                                ;ADDRESS
        LD      A,(HL)          ;GET CONTENTS OF
                                ;THAT ADDRESS AND PLACE
                                ;IN A REGISTER
        RET                     ;RETURN
```

BRKOUT deals with the ball colliding with an object. It determines which coloured brick (if any) it has hit and gives an appropriate score. When the ball hits a brick the variable HITS is deducted by one to keep a count of the number of bricks still left standing. It branches off to NOENO if any are still left.

If all the bricks are knocked down then the routine will reset the number of bricks by LOADing the variable HITS with 96 (i.e. three rows of 32 bricks). The player is rewarded by a bonus of two balls and the variable LEVEL is decreased. This controls the delay when moving the ball, thus increasing its speed for the next game.

```
BRKOUT:
        ;BRICK HAS HIT SOMETHING DO A TEST

        LD      BC,0
        CP      30H             ;HAVE WE HIT A YELLOW BRICK
        JR      NZ,NTYLW        ;NOT YELLOW
```

```
        LD      A,-1            ;SEND BALL IN OTHER DIRECTION
        LD      (YINC),A
        LD      BC,2            ;ADD TO SCORE
        JR      BEEP            ;AND MAKE A NOISE ABOUT IT!

NTYLW:
        CP      18H             ;HAVE WE HIT A
                                ;MAGENTA BRICK?
        JR      NZ,NTMAGN       ;NOT MAGENTA
        LD      A,-1            ;SEND BALL IN
                                ;OTHER DIRECTION
        LD      (YINC),A
        LD      BC,5            ;SCORE
        JR      BEEP            ;AND MAKE A SOUND

NTMAGN:
        CP      20H             ;HAVE WE HIT A GREEN BRICK?
        JR      NZ,ERROR        ;GOD KNOWS WHAT WE HIT!
        LD      BC,10           ;GIVE HIM A BIG SCORE
        JR      BEEP            ;AND MAKE A NOISE!

ERROR:
        LD      DE,40H
        LD      HL,666H
        CALL    3B5H            ;MAKE A LONGER BEEP!

BEEP:
        LD      HL,(SCORE)      ;GET SCORE
        ADD     HL,BC           ;AND ADD 0,5 OR 16
        LD      (SCORE),HL      ;SAVE UPDATED SCORE
        LD      HL,HITS         ;POINT TO NUMBER OF HITS
        DEC     (HL)            ;SUBTRACT ONE
        JR      NZ,NOEND        ;ALL BRICKS HIT?

        LD      (HL),96         ;RESET NUMBER OF BRICKS
        LD      A,(LEVEL)       ;GET LEVEL
        AND     A               ;TEST FOR ZERO LEVEL
        JR      Z,MAXLEV        ;DO NOT BOTHER MAKING
                                ;ANY MORE DIFFICULT
        DEC     A               ;ONE OFF THE LEVEL
        LD      (LEVEL),A       ;AND SAVE
        LD      A,(BALLS)       ;GET NUMBER OF BALLS
        ADD     A,2             ;AND GIVE HIM TWO MORE
        LD      (BALLS),A       ;AND SAVE
```

```
MAXLEV:
        CALL    RNDBAL          ;GET A RANDOM BALL POSTION
        CALL    SETUP           ;SET UP THE WALL
NOEND:
        LD      DE,8
        LD      HL,666H
        CALL    3B5H            ;BEEP
        LD      A,0             ;MAKE SURE
        OUT     (0FEH),A        ;WE HAVE A BLACK BORDER
        RET
```

The routine MUBALL is one of the main routines which deals with the movement of the ball. The ball has a Y direction (YINC) and X direction (XINC). These two variables are offsets which are added to the ball's X and Y co-ordinates. These are either 1 or −1. If the ball passes the bottom of the screen, one is deducted off the number of remaining balls. If there is still any left then a branch is made to RNDBALL which sets up another ball at a random X position. If the ball collides with an object than its X direction and/or Y direction is reversed.

```
MUBALL:
        LD      HL,(BALLX)      ;GET BALLS X,Y CO-ORD
        ;BALL DOESNT GO THROUGH
        ;THE BAT...

        LD      A,(YINC)        ;GET Y DIRECTION
        ADD     A,H             ;ADD TO Y
        LD      H,A             ;AND SAVE NEW Y CO-ORD
        LD      A,(XINC)        ;GET X DIRECTION
        ADD     A,L             ;ADD TO CURRENT X CO-ORD
        LD      L,A             ;AND SAVE NEW X CO-ORD
        PUSH    HL              ;SAVE THIS
        CALL    PEEK            ;LOOK AT THE COLOUR
                                ;OF NEW X,Y
        POP     HL              ;RESTORE SCREEN ADDRESS
        CP      39H             ;IS IT THE BAT
        ;HIT?
        JR      NZ,NTBAT        ;NO ITS NOT!

        LD      A,(YINC)        ;REVERSE Y DIRECTION
        NEG
        LD      (YINC),A        ;AND SAVE
        ;SEMI REBOUND DIRN.
```

```
        LD      A,(XINC)        ;REVERSE X DIRECTION
        NEG
        ADD     A,L             ;ADD X CO-ORD
        LD      L,A             ;AND SAVE IN L
        CALL    PEEK            ;LOOK AT COLOURS HERE
        CP      38H             ;IS IT NORMAL BACKGROUND
        JR      NZ,MUBALL       ;NO,THEN MOVE  BALL

        ;SEND BALL BACK THE WAY IT CAME ...

        LD      A,(BALLX)       ;GET X CO-ORD
        AND     A               ;TEST FOR LEFT HAND SIDE
        JR      Z,MUBALL        ;NO THEN MOVE BALL
        LD      A,(XINC)
        NEG                     ;REVERSE X DIRECTION
        LD      (XINC),A        ;AND SAVE
        JR      MUBALL          ;MOVE THE BALL

NTBAT:
        LD      HL,(BALLX)      ;GET X,Y CO-ORD
        LD      A,(XINC)        ;GET X DIRECTION
        ADD     A,L             ;GET NEW X CO-ORD
        LD      L,A             ;AND SAVE IN L REG
        LD      (TPBLYX),A      ;PLACE NEW X CO-ORD IN TEMP
        AND     A               ;IS IT AT THE
                                ;LEFT HAND SIDE?
        JR      Z,NGXINC        ;YES THEN GO TO
                                ;CHANGE X DIRECTION
        CP      1FH             ;IS IT ON THE
                                ;RIGHT HAND SIDE?
        JR      C,YCHECK        ;NO SO CHECK Y MOVEMENT
NGXINC:
        LD      A,(XINC)        ;GET X DIRECTION
        NEG                     ;REVERSE X DIRECTION
        LD      (XINC),A        ;AND SAVE

YCHECK:
        LD      A,(YINC)        ;GET Y DIRECTION
        ADD     A,H             ;GET NEW Y CO-ORD
        LD      H,A             ;AND SAVE IN H REG
        LD      (TPBLYX+1),A    ;AS WELL AS TEMP+1
        AND     A               ;HAVE WE HIT THE TOP?
        JR      Z,NGYINC        ;YES THEN CHANGE
                                ;Y DIRECTION
```

```
        CP      23              ;HAVE WE HIT THE BOTTOM?
        JR      NC,BALOUT       ;OUT OF BOUNDS
        CALL    PEEK            ;NO,LOOK AT WHERE
                                ;WE ARE GOING TO.
        CP      38H             ;IS IT BLANK?
        JR      Z,GOED          ;YES,CARRY ON
        CALL    BRKOUT          ;HIT SOMETHING SO CHECK!
NGYINC:
        LD      A,(YINC)        ;GET Y DIRECTION
        NEG                     ;CHANGE DIRETCION
        LD      (YINC),A        ;AND SAVE
GOED:
        LD      A,0             ;SET UP CHARACTER AS SPACE
        ;ERASE OLD
        LD      C,38H           ;SET UP BACKGROUND COLOUR
        LD      HL,(BALLX)      ;GET BALLS X,Y CO-ORDS
        CALL    PRTCH           ;AND BLANK OUT
        LD      A,1             ;GET BALL CHARACTER
        LD      HL,(TPBLYX)     ;GET TEMP X,Y
        LD      (BALLX),HL      ;AND SAVE IN BALLS
                                ;X,Y CO-ORDS
        CALL    PRTCH           ;PRINT THE BALL!
        LD      A,(LEVEL)       ;GET LEVEL
LEVLP:
        EI
        HALT                    ;1/50 OF A SECOND DELAY
        DI
        DEC A
        JR      NZ,LEVLP        ;DELAY DEPENDENT ON LEVEL
        RET
BALOUT:
        LD      A,0             ;SET UP CHARACTER AS SPACE
        ;ERASE BALL             ;BACKGROUND COLOUR
        LD      C,38H
        LD      HL,(BALLX)      ;GET X,Y CO-ORDS
        CALL    PRTCH           ;AND PRINT
        CALL    RNDBAL          ;GET RANDOM X,Y CO-ORD
        LD      HL,BALLS        ;ONE OFF NUMBER OF BALLS
        DEC     (HL)
        RET

;RANDOM X-POSN FOR BALL
```

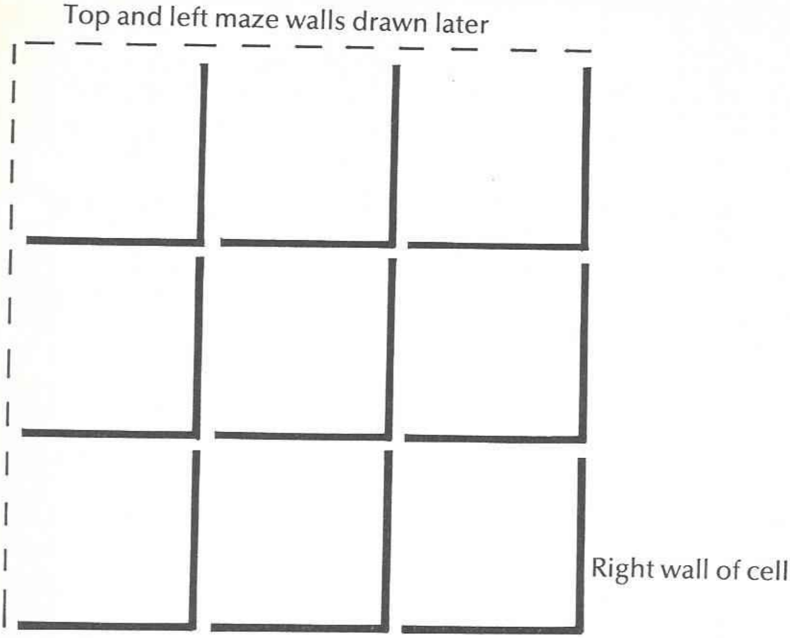
```
RNDBAL:                               ;GET FRAMES
        LD      A,(23672)
        ;LSB OF FRAMES
        SRL     A
        AND     0FH                   ;0-15
        ADD     A,5                   ;5-20
        LD      (BALLX),A             ;SAVE RANDOM X CO-ORD
                                      ;FOR BALL
        LD      A,6                   ;INTIALIZE Y CO-ORD
        ;SET Y -POS
        LD      (BALLX+1),A           ;AND SAVE
        LD      HL,(BALLX)            ;GET X,Y CO-ORD
        LD      (TPBLYX),HL           ;AND SAVE IN TEMP X,Y
        LD      B,50
DLOOP:
        EI
        HALT
        DI
        DJNZ    DLOOP                 ;WAIT FOR A WHILE
        RET


        END
```

MAZE GENERATOR

This program generates random mazes consisting of a 32 by 22 grid of cells. It can, however, be easily adapted to produce mazes of any desired width and height. The algorithm used generates mazes where there is only one route from one cell to another. This routine could be used in games to produce a maze for an adventure game program such as the famous 'Hall of the Things' by Crystal Software.

A maze is constructed of cells and each of these cells are surrounded by up to four walls. They can be regarded as having just two walls as the other surrounding cells provide the other two. Diagram 1 shows how the wall would be made up on a 3 by 3 maze. Notice that there are no walls on the top and left sides of the maze. These we can draw later, after we have constructed the rest of the maze.



Hexadecimal Listing

```
7D00    C3  40  7D  0F  16  10  01  00
7D08    00  01  01  04  00  00  00  00
7D10    00  00  00  00  00  00  00  00
7D18    3C  7E  FF  FF  FF  FF  7E  3C
7D20    3F  7F  FF  FF  FF  FF  7F  3F
7D28    FF  FF  FF  FF  FF  FF  FF  FF
7D30    FC  FE  FF  FF  FF  FF  FE  FC
7D38    FF  81  81  81  81  81  81  FF

7D40    3E  02  CD  01  16  AF  D3  FE
7D48    21  00  00  22  0C  7D  3E  05
7D50    32  0E  7D  3E  04  32  0B  7D
7D58    3E  60  32  0F  7D  CD  1A  7E
7D60    CD  5F  7F  21  10  16  22  03
7D68    7D  CD  D9  7D  3A  0E  7D  A7
7D70    28  0B  CD  83  7D  CD  B7  7E
7D78    FB  76  F3  18  EF  ED  4B  0C
```

7D80	7D	FB	C9	CD	D9	7D	3E	FE
7D88	DB	FE	E6	1F	FE	1F	C8	CD
7D90	C5	7D	CB	4F	CC	B7	7D	CB
7D98	57	CC	A8	7D	CB	47	20	04
7DA0	CB	C7	18	EE	CD	D9	7D	C9
7DA8	F5	2A	03	7D	3E	1D	BD	28
7DB0	04	2C	22	03	7D	F1	C9	F5
7DB8	2A	03	7D	7D	A7	28	04	2D
7DC0	22	03	7D	F1	C9	F5	2A	03
7DC8	7D	01	38	03	C5	E5	AF	CD
7DD0	ED	7D	E1	2C	C1	10	F5	F1
7DD8	C9	2A	03	7D	01	39	03	3E
7DE0	02	C5	E5	CD	ED	7D	3C	E1
7DE8	2C	C1	10	F5	C9	F5	C5	E5
7DF0	C5	F5	E5	7C	CD	9E	0E	D1
7DF8	16	00	19	EB	F1	01	10	7D
7E00	26	00	6F	29	29	29	09	06
7E08	08	7E	12	23	14	10	FA	EB
7E10	CD	88	0E	C1	79	12	E1	C1
7E18	F1	C9	CD	6B	0D	01	20	20
7E20	3E	05	21	00	03	CD	37	7E
7E28	01	18	20	21	00	04	CD	37
7E30	7E	01	30	20	21	00	05	CD
7E38	ED	7D	2C	10	FA	C9	7D	6C
7E40	26	00	29	29	29	29	29	06
7E48	00	4F	09	01	00	58	09	7E
7E50	C9	01	00	00	FE	30	20	0A
7E58	3E	FF	32	0A	7D	01	02	00
7E60	18	20	FE	18	20	0A	3E	FF
7E68	32	0A	7D	01	05	00	18	12
7E70	FE	20	20	05	01	0A	00	18
7E78	09	11	40	00	21	66	06	CD
7E80	B5	03	2A	0C	7D	09	22	0C
7E88	7D	21	0F	7D	35	20	1A	36
7E90	60	3A	0B	7D	A7	28	0C	3D
7E98	32	0B	7D	3A	0E	7D	C6	02
7EA0	32	0E	7D	CD	5F	7F	CD	1A
7EA8	7E	11	08	00	21	66	06	CD
7EB0	B5	03	3E	00	D3	FE	C9	2A
7EB8	05	7D	3A	0A	7D	84	67	3A

```
7EC0   09  7D  85  6F  E5  CD  3E  7E
7EC8   E1  FE  39  20  26  3A  0A  7D
7ED0   ED  44  32  0A  7D  3A  09  7D
7ED8   ED  44  85  6F  CD  3E  7E  FE
7EE0   38  20  D4  3A  05  7D  A7  28
7EE8   CE  3A  09  7D  ED  44  32  09
7EF0   7D  18  C4  2A  05  7D  3A  09
7EF8   7D  85  6F  32  07  7D  A7  28

7F00   04  FE  1F  38  08  3A  09  7D
7F08   ED  44  32  09  7D  3A  0A  7D
7F10   84  67  32  08  7D  A7  28  0E
7F18   FE  17  30  31  CD  3E  7E  FE
7F20   38  28  0B  CD  51  7E  3A  0A
7F28   7D  ED  44  32  0A  7D  3E  00
7F30   0E  38  2A  05  7D  CD  ED  7D
7F38   3E  01  2A  07  7D  22  05  7D

7F40   CD  ED  7D  3A  0B  7D  FB  76
7F48   F3  3D  20  FA  C9  3E  00  0E
7F50   38  2A  05  7D  CD  ED  7D  CD
7F58   5F  7F  21  0E  7D  35  C9  3A
7F60   78  5C  CB  3F  E6  0F  C6  05
7F68   32  05  7D  3E  06  32  06  7D
7F70   2A  05  7D  22  07  7D  06  32
7F78   FB  76  F3  10  FB  C9  A7  28
```

The theory behind the maze generator is to walk randomly round the maze knocking down walls as we proceed. To begin we walk a set number of steps around the maze knocking down the walls if we meet any obstructions. On our second walk we start off in a cell that we have not previously entered. We again walk randomly around the maze knocking walls down. We keep on walking till we arrive at a cell which we had visited before on 'other walks'. When we arrive at such a cell we have then completed a path from one random walk to another on a different random walk. This process is repeated on all the untouched cells until we have proceeded through all the maze.

To implement this algorithm in machine code we represent our maze by having two arrays, the size of which are the size of the number of cells in the maze. One is called BUILD, the other MAZE. The array BUILD holds the path numbers and route which we 'walk' along while the array MAZE holds the 'wall' patterns. A wall pattern shows the structure of the two walls in a cell. The array MAZE is initialized

with the two walls intact. This is represented by the two first bits of its number being set high (i.e. the number three). Knocking down the walls is represented by re-setting a particular bit. If bit 0 of the number represents the bottom wall and bit 1 represents the right hand side wall then we can see the process if we knock down a wall. Going downwards we reset bit 0 of the cell we are in. If we knock down a wall going up we reset bit 0 of the cell above,the cell we are entering. Going right we re-set bit 1 of the cell we are in, going left we re-set bit 1 of the adjacent cell.

One point we have to look out for is that we do not 'back track' on a particular walk we are doing. We do this by giving each walk a path number and if we do happen to back track on our original path then we do not bother to knock down any walls. Using this method we guarantee our maze does not have any gaping holes and that it is singular in nature.

The program comes in two parts, one BASIC and one machine code. The machine code routine generates a random maze. The BASIC program draws the top and left hand side of the wall to complete the maze. When you use the generator in a game the unused bits in the array MAZE can be used to represent up to 63 objects such as axes, torches, wands or nasties! The second array is unused once the maze is generated so it could be used to store other variables or data in the game. The maze takes about two seconds to generate, very slow by machine code standards. perhaps you could set yourself the task to make it faster. One way of improving the speed for 48K Spectrum owners would be to place the routine higher up in the memory map above the address 32768. Moving the code here would stop the Z80 CPU 'waiting' for the Spectrum's ULA to update the screen.

BASIC Listing

```
   1 CLEAR 29000
  10 CLS:LET SC=USR 32000
  20 CLS:PRINT AT 10,10;"SCORE =
 ";SC
  30 FOR X=1 TO 200:NEXT X
  40 PRINT #0;"PRESS A KEY TO ST
ART"
  50 PAUSE 0
  60 GO TO 10
```

Assembler Listing

```
        ORG     30000
        JP      START
XPOS:   DB      0
YPOS:   DB      0
PATH:   DB      0

OPENCH  EQU     1601H
UDG     EQU     23675           ;UDG ADDRESS

INTMAZE:
        LD      A,2             ;OPEN SCREEN CHANNEL
        CALL    OPENCH
        LD      HL,NOUGHT       ;SET USER DEF GRAPHICS
                                ;TO OURS
        LD      (UDG),HL
        CALL    RANDI           ;INTIALIZE RANDOM
                                ;NUMBER GENERATOR
        LD      HL,MAZE         ;RE-BUILD THE MAZE
        LD      DE,MAZE+1
        LD      BC,22*32        ;OF 22 BY 32
        LD      (HL),3          ;WITH WALLS
        LDIR
;   HL POINTS TO BUILD
;   DE POINTS TO BUILD+1

        LD      BC,22*32        ;CLEAR THE ARRAY
        LD      (HL),0          ;BUILD WITH 0
        LDIR
        XOR     A               ;SET THE START X CO-ORD
        LD      (XPOS),A
        LD      (YPOS),A        ;SET THE START Y CO-ORD
        INC     A
        LD      (PATH),A        ;SET THE STARTING
                                ;PATH NUMBER
        RET
                                ;FINISH INTIALIZING


RAND0:  DB      0               ;RANDOM VAR 0
RAND1:  DB      0               ;RANDOM VAR 1
```

```
RAND2:  DB      0               ;RANDOM VAR 2
RAND3:  DB      0               ;RANDOM VAR 3

RAND:                           ;GENERATE RANDOM NUMBER
                                ;BETWEEN
                                ;0 AND 255

        LD      A,(RAND1)       ;GET RANDOM SEED
        RRCA                    ;A MOD 8 * 32
        RRCA
        RRCA
        PUSH    BC              ;SAVE BC PAIR
        PUSH    AF              ;SAVE AF PAIR
        LD      A,(RAND2)       ;GET SECOND RANDOM VARIABLE
        LD      B,A             ;AND PLACE IN B REGISTER
        LD      A,(RAND3)       ;GET THIRD RANDOM VARIABLE
        LD      C,A             ;AND PLACE IN C REGISTER
        POP     AF              ;RESTORE AF
        ADD     A,B             ;(RAND1)MOD 8 *32 + (RAND2)
        ADD     A,C             ;+(RAND3)
        RLCA                    ;ALIGN  BITS
        RLCA                    ;AND SAVE NEW RANDOM
                                ;VARIABLES
        LD      (RAND0),A
        LD      A,B
        LD      (RAND1),A
        LD      A,C
        LD      (RAND2),A
        LD      A,(RAND0)
        LD      (RAND3),A
        POP     BC              ;RESTORE BC PAIR
        RET

RANDI:
        LD      A,0             ;SET UP RANDOM VARIABLES
        LD      (RAND0),A
        LD      A,173
        LD      (RAND1),A
        LD      A,206
        LD      (RAND2),A
        LD      A,R             ;ENSURE SOME RANDOMNESS
        LD      (RAND3),A
        RET
```

```
LENW    EQU     255D            ;LARGEST WALK

WALK:
        LD      B,LENW
KEW:                            ;KEEP WALKING

        CALL    RANDW
        LD      A,(HL)          ;GET CONTENTS OF
                                ;NEW POSTION
        AND     A               ;TEST FOR NEW LOCATION
        JR      Z,PUTIN         ;ZERO SO MARK PATH!
        CP      C               ;GOING BACK ON PATH?
        JR      Z,PUTIN         ;YES MARK IT!
        RET                     ;HAVE REACHED A VALUE LOWER

PUTIN:
        LD      A,(PATH)        ;GET PATH NUMBER
        LD      (HL),A          ;AND PLACE IN BUILD
        DJNZ    KEW             ;DO THIS FOR LENW MAXIMUM
        LD      A,(PATH)        ;ONLY DO LENW FOR PATH 1
        CP      1
        JR      NZ,WALK

        RET



RANDW:
        CALL    RAND            ;GET RANDOM NUMBER
        AND     3               ;MASK OFF FOR
                                ;NUMBERS 0 TO 3
        AND     A


        JR      Z,NORTH         ;IF ZERO GO NORTH
        CP      1
        JR      Z,SOUTH         ;IF 1 GO SOUTH
        CP      2
        JR      Z,WEST          ;IF 2 GO WEST

;GO EAST
```

```
EAST:
        LD      A,(XPOS)        ;GET X CO-ORD
        CP      31              ;TEST TO SEE IF WE ARE ON
        JR      Z,RANDW         ;THE RIGHT HAND SIDE
                                ;IF SO GO AGAIN
        INC     A               ;ELSE INCREASE
                                ;X CO-ORD BY 1
        LD      (XPOS),A        ;AND SAVE
        LD      C,(HL)          ;OLD VALUE  IN C REGISTER
        INC     HL              ;NEW POSTION
        LD      A,(HL)          ;NEW VALUE
        CP      C               ;ARE THEY EQUAL?
        RET     Z               ;DON'T BACKTRACK!

        DEC     HL              ;GET OLD POSTION
        CALL    RES1            ;RESET BIT 1 OF OLD CELL
        INC     HL              ;POINT TO NEW CELL
        RET




WEST:

        LD      A,(XPOS)        ;GET X CO-ORD
        AND     A               ;TEST FOR LEFT HAND SIDE
        JR      Z,RANDW         ;PICK ANOTHER DIRECTION


        DEC     A               ;GO LEFT
        LD      (XPOS),A        ;SAVE X CO-ORD
        LD      C,(HL)          ;OLD VALUE


        DEC     HL              ;GO LEFT
        LD      A,(HL)          ;GET NEW VALUE
        CP      C               ;COMPARE WITH OLD VALUE
        RET     Z               ;NO BACKTRACKING
```

```
RES1:   PUSH    HL              ;SAVE NEW POSTION
        LD      DE,MAZE-BUILD   ;POINT TO CORRESPONDING
                                ;MAZE ADDR
        ADD     HL,DE
        RES     1,(HL)          ;KNOCK DOWN WALL
        POP     HL              ;GET NEW POSTION
        RET


NORTH:
        LD      A,(YPOS)        ;GET Y CO-ORD
        AND     A               ;ARE WE AT THE TOP?
        JR      Z,RANDW         ;YES, THEN PICK ANOTHER
                                ;DIRECTION


        DEC     A               ;GOING UP
        LD      (YPOS),A        ;SAVE NEW Y CO-ORD
        LD      C,(HL)          ;GET OLD VALUE
        LD      DE,-32          ;OFFSET FOR GOING UP
        ADD     HL,DE           ;POINT TO NEW PART OF
                                ;BUILD ARRAY
        LD      A,(HL)          ;GET PATH NUMBER
        CP      C               ;COMPARE WITH OLD VALUE
        RET     Z               ;DON'T BACKTRACK!


RES0:   PUSH    HL              ;SAVE NEW POSTION
        LD      DE,MAZE-BUILD   ;POINT TO CORRESPONDING
                                ;MAZE
        ADD     HL,DE
        RES     0,(HL)          ;KNOWN DOWN WALL
        POP     HL              ;RESTORE POSTION
        RET


SOUTH:
        LD      A,(YPOS)        ;GET Y CO-ORD
        CP      21              ;HAVE WE HIT THE BOTTOM?
        JR      Z,RANDW         ;YES THEN PICK
                                ;ANOTHER DIRECTION
```

```
        INC     A               ;GOING DOWN
        LD      (YPOS),A        ;SAVE Y CO-ORD
        LD      C,(HL)          ;OLD VALUE.

        LD      DE,32           ;OFFSET FOR GOING DOWN
        ADD     HL,DE
        LD      A,(HL)          ;GET NEW PATH NUMBER
        CP      C               ;COMPARE WITH OLD PATH
        RET     Z               ;DON'T BACKTRACK
        AND     A               ;CLEAR CARRY
        SBC     HL,DE           ;NORMAL SUBTRACTION
        CALL    RES0            ;KNOCK DOWN BOTTOM WALL
        LD      DE,32           ;GET HL BACK
        ADD     HL,DE
        RET

PRINTC:
        PUSH    HL              ;SAVE REGISTERS
        PUSH    BC


        RST     010H            ;PRINT A REGISTER
                                ;TO CURRENT CHANNAL
                                ;RESTORE REGISTERS
        POP     BC
        POP     HL
        RET



DISPLAY:                        ;DISPLAY MAZE TO
        LD      HL,MAZE         ;SCREEN.LOAD A WITH
        LD      A,22
                                ;NUMBER OF LINES DOWN
LINE:   PUSH    AF              ;SAVE LINE COUNT

        LD      B,32            ;GET CHARACTER COUNT
DRAWS:                          ;GET MAZE VALUE
        LD      A,(HL)
        ADD     A,144           ;ADD BASE OF UDG
        CALL    PRINTC          ;PRINT CHARACTER
        INC     HL              ;NEXT MAZE CELL
        DJNZ    DRAWS           ;REPEAT 32 TIMES
```

```
        POP     AF              ;RESTORE LINE NUMBER
        DEC     A               ;ONE OFF LINE NUMBER
        JR      NZ,LINE         ;REPEAT 22 TIME
        RET



BUILDM:
        LD      HL,BUILD        ;POINT TO ARRAY BUILD
        XOR     A               ;INTIALIZE THE X CO-ORD
        LD      (XPOS),A        ;AND Y CO-ORD
        LD      (YPOS),A

FIND:
        LD      A,(HL)          ;GET PATH NUMBER
        AND     A               ;TEST FOR ZERO
        JR      Z,SKIP          ;START WALKING ON ZERO
        INC     HL              ;NEXT ONE ACROSS
        LD      A,(XPOS)        ;UPDATE X CO-ORD
        INC     A
        LD      (XPOS),A
        CP      32              ;HAVE WE GONE RIGHT ACROSS?
        JR      NZ,FIND         ;NO, SO CARRY ON LOOKING
        XOR     A               ;YES...RESET X CO-ORD
        LD      (XPOS),A
        LD      A,(YPOS)        ;GO ONE DOWN
        INC     A
        LD      (YPOS),A
        CP      22              ;HAVE WE GONE ALL
        JR      NZ,FIND         ;THE WAY DOWN?.NO THEN KEEP
        RET                     ;LOOKING..ELSE RETURN

SKIP:
        LD      A,(PATH)        ;GET PATH NUMBER
        LD      (HL),A          ;PLACE AT NEW CELL
        CALL    WALK            ;DO A RANDOM WALK
        LD      A,(PATH)        ;UPDATE.....
        INC     A
        LD      (PATH),A        ;NEW PATH NUMBER
        JR      BUILDM          ;CARRY ON BUILDING
```

```
START:
        CALL    INTMAZE         ;CLEAR MAZE
        CALL    BUILDM          ;BUILD MAZE
        CALL    DISPLAY         ;DISPLAY MAZE
        RET

;USER DEFINE CHARACTER SET FOR MAZE

NOUGHT:
        DB      0,0,0,0,0,0,0,0
        DB      0,0,0,0,0,0,0,255
        DB      1,1,1,1,1,1,1,1
        DB      1,1,1,1,1,1,1,255



MAZE:   DS      32*22
BUILD:  DS      32*22

        END
```

Hexadecimal Listing

```
7530    C3  8D  76  00  00  00  3E  02
7538    CD  01  16  21  97  76  22  7B
7540    5C  CD  90  75  21  B7  76  11
7548    B8  76  01  C0  02  36  03  ED
7550    B0  01  C0  02  36  00  ED  B0
7558    AF  32  33  75  32  34  75  3C
7560    32  35  75  C9  00  00  00  00
7568    3A  65  75  0F  0F  0F  C5  F5

7570    3A  66  75  47  3A  67  75  4F
7578    F1  80  81  07  07  32  64  75
7580    78  32  65  75  79  32  66  75
7588    3A  64  75  32  67  75  C1  C9
7590    3E  00  32  64  75  3E  AD  32
7598    65  75  3E  CE  32  66  75  ED
75A0    5F  32  67  75  C9  06  FF  CD
75A8    C0  75  7E  A7  28  04  B9  28
```

75B0	01	C9	3A	35	75	77	10	EF
75B8	3A	35	75	FE	01	20	E6	C9
75C0	CD	68	75	E6	03	A7	28	36
75C8	FE	01	28	4D	FE	02	28	16
75D0	3A	33	75	FE	1F	28	E9	3C
75D8	32	33	75	4E	23	7E	B9	C8
75E0	2B	CD	F5	75	23	C9	3A	33
75E8	75	A7	28	D4	3D	32	33	75
75F0	4E	2B	7E	B9	C8	E5	11	40
75F8	FD	19	CB	8E	E1	C9	3A	34
7600	75	A7	28	BC	3D	32	34	75
7608	4E	11	E0	FF	19	7E	B9	C8
7610	E5	11	40	FD	19	CB	86	E1
7618	C9	3A	34	75	FE	15	28	A0
7620	3C	32	34	75	4E	11	20	00
7628	19	7E	B9	C8	A7	ED	52	CD
7630	10	76	11	20	00	19	C9	E5
7638	C5	D7	C1	E1	C9	21	B7	76
7640	3E	16	F5	06	20	7E	C6	90
7648	CD	37	76	23	10	F7	F1	3D
7650	20	F0	C9	21	77	79	AF	32
7658	33	75	32	34	75	7E	A7	28
7660	1C	23	3A	33	75	3C	32	33
7668	75	FE	20	20	F0	AF	32	33
7670	75	3A	34	75	3C	32	34	75
7678	FE	16	20	E1	C9	3A	35	75
7680	77	CD	A5	75	3A	35	75	3C
7688	32	35	75	18	C6	CD	36	75
7690	CD	53	76	CD	3D	76	C9	00
7698	00	00	00	00	00	00	00	00
76A0	00	00	00	00	00	00	FF	01
76A8	01	01	01	01	01	01	01	01
76B0	01	01	01	01	01	01	FF	00
76B8	00	00	00	00	00	00	00	00
76C0	00	00	00	00	00	00	00	00
76C8	00	00	00	00	00	00	00	00
76D0	00	00	00	00	00	00	00	00
76D8	00	00	00	00	00	00	00	00
76E0	00	00	00	00	00	00	00	00
76E8	00	00	00	00	00	00	00	00

```
76F0   00  00  00  00  00  00  00  00
76F8   00  00  00  00  00  00  00  00
7700   00  00  00  00  00  00  00  00
7708   00  00  00  00  00  00  00  00
7710   00  00  00  00  00  00  00  00
7718   00  00  00  00  00  00  00  00
7720   00  00  00  00  00  00  00  00
7728   00  00  00  00  00  00  00  00
```

LARGE PRINT

I wrote this routine to enhance my own BASIC programs. The routine PRINTS characters on the screen twice the width of normal characters. I have 'patched' part of the BASIC operating system so that the large characters can be PRINTed from BASIC and will accept all the control characters, such as INK, PAPER, AT, TAB, etc. To enable the large PRINT facility we first call the routine at address 30000. This gives the Spectrum an additional channel, channel number 5. Then, to PRINT large characters to the screen we simply us the BASIC syntax:

PRINT#5;"STRING"

Here's a sample BASIC program which demonstrates how the routine can be used:

```
  10 CLS : RANDOMIZE USR 30000
  20 PRINT "This program demonst
rates"
  30 PRINT "How to get ";: PRINT
 #5;"Large";: PRINT " letters"
  40 PRINT #5;"    on the screen
"
  50 PRINT
  60 PRINT "It can cope with con
trol codes"
  70 PRINT #5;AT 5,5;"such as AT
"
  75 PRINT
  80 PRINT #5; INK 5; PAPER 2;"a
nd colours"
  90 PRINT #5; INVERSE 1;TAB 7;"
inverse"
 100 PRINT #5; FLASH 1;" as well
 as flashing"
```

Assembler Listing

```
        ORG     30000

CURCHL  EQU     23633
REPORTJ EQU     15C4H           ;INVALID I/O DEVICE
STRMS           EQU     23568D  ;STREAMS
STREAM5 EQU     STRMS+6+5*2     ;OPEN   CHANNEL 5



CHARS   EQU     23606D          ;CHARS CHARACTER ADDRESS
UDG     EQU     23675D          ;UDG ADDRESS
CHANS   EQU     23631D          ;CHANNEL ADDRESS
CHANINF EQU     STREAM5+2       ;CHANNEL 5


POCHANGE        EQU     0A80H
TVDATA  EQU     23566
TVDATL  EQU     TVDATA
TVDATH  EQU     TVDATA+1
POCONT  EQU     0A87H

INITP:                          ;SET UP PRINT#4 COMMAND
        LD      HL,CHANIND      ;MOVE CHANNEL INFORMATION
        LD      DE,CHANINF
        LD      BC,5
        LDIR
        LD      HL,CHANINF      ;FIND DISTANCE BETWEEN CHAN
        LD      DE,(CHANS)
        AND     A
        SBC     HL,DE
        INC     HL
        LD      (STREAM5),HL    ;SET UP STREAMS
        RET

CHANIND:
        DEFW    PRINTD          ;PRINT OUT ROUTINE
        DEFW    REPORTJ         ;INPUT ROUTINE.
        DEFB    'D'
```

```
PRINTD:
        ;WHEN BASIC CALLS THIS ROUTINE THE
        ;A REG CONTAINS CHAR NUMBER.

        CP      20H             ;TEST TO SEE IF PRINTABLE
        JP      NC,CAR          ;PRINT THE CHARACTER
        CALL    0B03H           ;GET CURRENT PRINT POSTION
        CP      06              ;PRINT '?'
                                ;FOR CODES 00- 05 HEX
        JP      C,0A69H
        CP      18H             ;AND 18H TO 1FH
        JP      NC,0A69H
        CP      16
        JP      C,09F4H+16D     ;GO TO ROMS TABLE.

        LD      HL,ATTAB        ;ASSUME CONTROL CHAR
                                ;IS AT OR TAB
        CP      22
        JR      NC,RIGHT        ;YOU WHERE RIGHT!
        LD      HL,INKOVER
RIGHT:
        PUSH    HL              ;RETURN ADDRESS IS PUSHED
        JP      0B03H           ;FETCH CURRENT CHARACTER


POTV2D:
        LD      DE,POCONTD      ;SAVE FIRST OPERAND
                                ;IN TVDATH
        LD      (TVDATH),A
        JP      POCHANGE        ;CHANGE ADDRESS OF
                                ;CURRENT CHANNEL


ATTAB:
        LD      DE,POTV2D       ;NEXT TIME ROUND GOTO POTVD
        JR      POTV1D          ;SAVE CHARACTER CODE
                                ;IN TVDATAL

INKOVER:
        LD      DE,POCONTD      ;NEXT TIME POCONTD
POTV1D: LD      (TVDATL),A      ;SAVE CONTROL CODE
        JP      POCHANGE        ;CHANGE OUTPUT ADDRESS
```

```
POCONTD:                        ;BOTH OPERANDS ARE
                                ;COLLECTED
        LD      DE,PRINTD       ;NEXT TIME PRINTD
        JP      POCONT+3        ;DEAL WITH OP AND CONT CODE




CAR:    LD      DE,(CHARS)      ;GET CHARACTER ADDRESS

        LD      H,0             ;GET CHARACTER CODE IN HL
        LD      L,A
        ADD     HL,HL           ;*2
        ADD     HL,HL           ;*4
        ADD     HL,HL           ;*8
        ADD     HL,DE           ;HL POINTS TO START OF
                                ;CHAR DATA
        EX      DE,HL           ;NOW DE DOES!

        LD      HL,DUGD
        PUSH    HL
        POP     IX              ;EX HL WITH IX REG
        LD      B,8             ;DATA COUNT

GETD:   LD      A,(DE)          ;GET DATA

;       FECTHD              GET DOUBLE WORD IN HL

        PUSH    BC              ;SAVE DATA COUNT
        CALL    FETD
        POP     BC              ;RESTORE DATA COUNT

        LD      (IX+0),L        ;PLACE IN UDG AREA
        LD      (IX+8),H
        INC     IX              ;NEXT BYTE IN UDG
        INC     DE              ;NEXT CHAR DATA
        DJNZ    GETD            ;DO THIS EIGHT TIMES
        LD      HL,(UDG)        ;SAVE REAL UDG ADDRESS
        PUSH    HL
        LD      HL,DUGD         ;GET DUMMY UDG ADDRESS
        LD      (UDG),HL        ;AND CHANGE UDG
        LD      A,145           ;NOW PRINT USER DEFINED
                                ;GRAPHICS
```

```
        CALL    09F4H
        LD      A,144           ;NOW PRINT USER DEFINED
                                ;GRAPHICS
        CALL    09F4H
        POP     HL              ;RESTORE UDG
        LD      (UDG),HL
        RET

FETD:
        CALL    NYBBLE          ;DO ONE NYBBLE
        LD      L,H
;NOW GET NEXT NYBBLE
NYBBLE:
        LD      B,4             ;NUMBER OF BITS
NBIT:   RRCA

;TRY CHANGING ABOVE OPCODE TO RLCA!!! FOR A BIT OF FUN.

        RR      C               ;MAKE TWICE AS FAT
        SRA     C
        DJNZ    NBIT            ;DO 8 TIMES
        LD      H,C
        RET

DUGD:   DS      8*2

        END
```

Hexadecimal Listing

```
7530   21  4A  75  11  22  5C  01  05
7538   00  ED  B0  21  22  5C  ED  5B
7540   4F  5C  A7  ED  52  23  22  20
7548   5C  C9  4F  75  C4  15  44  FE
7550   20  D2  91  75  CD  03  0B  FE
7558   06  DA  69  0A  FE  18  D2  69
7560   0A  FE  10  DA  04  0A  21  7D
7568   75  FE  16  30  03  21  82  75
```

7570	E5	C3	03	0B	11	8B	75	32
7578	0F	5C	C3	80	0A	11	74	75
7580	18	03	11	8B	75	32	0E	5C
7588	C3	80	0A	11	4F	75	C3	8A
7590	0A	ED	5B	36	5C	26	00	6F
7598	29	29	29	19	EB	21	DE	75
75A0	E5	DD	E1	06	08	1A	C5	CD
75A8	CF	75	C1	DD	75	00	DD	74
75B0	08	DD	23	13	10	EF	2A	7B
75B8	5C	E5	21	DE	75	22	7B	5C
75C0	3E	91	CD	F4	09	3E	90	CD
75C8	F4	09	E1	22	7B	5C	C9	CD
75D0	D3	75	6C	06	04	0F	CB	19
75D8	CB	29	10	F9	61	C9	00	00
75E0	00	00	00	00	00	00	00	00
75E8	00	00	00	00	00	00	82	75
75F0	E5	C3	03	0B	11	8B	75	32
75F8	0F	5C	C3	80	0A	11	74	75

PIXEL SCROLL

This routine allows the user to scroll any portion of the screen to either left or right. It has a 'wrap-around' effect, and so could be most useful when writing arcade games with scrolling background scenery of mountains, high rise flats or the like. When calling the routine the HL register pair must point to the screen address of the position from which you wish to scroll. The program below is a demonstration program showing how the routine can be used from BASIC:

```
   5 FOR X=1 TO 32*22:PRINT CHR$
 143;:NEXT X
  10 PRINT AT 0,0;"THIS SCROLL W
ILL GO LEFT"
  20 PRINT "WITH THIS LINE!!"
  30 PRINT AT 8,0;"THIS SCROLL W
ILL GO RIGHT"
  40 PRINT "ALONG WITH THIS LINE
!!"
  50 RANDOMIZE USR 32000
  60 GO TO 50
```

Here are the listings for the scroll routine:

Assembler Listing

```
        ORG     32000D

NLINES  EQU     16              ;NUMBER OF LINES TO SCROLL
NBYTES  EQU     32              ;NUMBER OF BYTES TO SCROLL
ADD     EQU     16384           ;SCREEN ADDRESS
ADD2    EQU     16384+256*8     ;SCREEN ADDRESS2

        JP      TEST            ;TEST THE SCROLL

        ;********************************************
        ;These two routines SLEFT SCROLL LEFT
        ; and SRIGHT  SCROLL RIGHT scroll the screen
        ;left and right respectivly.They use the rout-
        ;-ine INCY which finds address of corresponding
        ;pixel line addresses.
        ;On entry to the routine HL points to the top
        ;left hand side of the portion of the screen
        ;to be scrolled.
        ;The other values which the program will give are
        ;NBYTES number of bytes to scroll  ie width
        ;NLINES number of lines to scroll
        ;Both rountines have a wrap-around effect.

SLEFT:
        LD      HL,ADD+NBYTES-1
                                ;POINT TO RIGHT HAND SIDE
        LD      C,NLINES        ;NUMBER OF LINES TO SCROLL

LINE:   PUSH    HL              ;SAVE SCREEN ADDRESS
        LD      B,NBYTES        ;NUMBER OF BYTES TO SCROLL
                                ;ACROSS

CHARX:
        RL      (HL)            ;SCROLL LEFT THROUGH
        DEC     HL              ;CARRY
        DJNZ    CHARX           ;REPEAT NBYTES TIMES
        POP     HL              ;RESTORE RIGHT HAND
                                ;SIDE ADDRESS
        LD      A,0             ;SET A TO ZERO
```

```
        ADC     A,A             ;PLACE CARRY IN BIT 0 OF
        OR      (HL)            ;RIGHT HAND SIDE
        LD      (HL),A
        CALL    INCY            ;GET ADDRESS OF NEXT PIXEL
                                ;LINE
        DEC     C               ;DOWN,ONE LESS LINE
        JR      NZ,LINE         ;REPEAT UNTIL DONE ALL LINES
        RET

SRIGHT:
        LD      HL,ADD2         ;POINT TO LEFT HAND
                                ;OF SCREEN
        LD      C,NLINES        ;NUMBER OF LINES TO SCROLL
LINER:  PUSH    HL              ;SAVE LEFT HAND
                                ;SIDE ADDRESS
        LD      B,NBYTES        ;NUMBER OF BYTES TO SCROLL

CHARR:
        RR      (HL)            ;SCROLL RIGHT THROUGH CARRY
        INC     HL              ;TO THE RIGHT
        DJNZ    CHARR           ;REPEAT UNTIL DONE
                                ;NBYTES TIMES
        POP     HL              ;RESTORE LEFT HAND SIDE
        LD      A,0             ;ROTATE CARRY INTO LEFT
                                ;HAND SIDE
        RRA
        OR      (HL)
        LD      (HL),A
        CALL    INCY            ;NEXT PIXEL LINE DOWN
        DEC     C               ;ONE LESS PIXEL LINE
        JR      NZ,LINER        ;REPEAT UNTIL NO MORE LINES
        RET


INCY:                           ;NEXT PIXEL LINE DOWN
                                ;HL POINTS TO SCREEN
                                ;ADDRESS
        INC     H               ;NEXT LINE DOWN
        LD      A,H             ;TEST IF WITHIN CHARACTER
        AND     7
        RET     NZ              ;WITHIN CHAR SO RETURN

        LD      A,L             ;NEXT CHARACTER DOWN
```

```
        ADD     A,20H
        LD      L,A
        RET     C               ;DO NOT ADJUST SECTOR SINCE
                                ;WE HAVE GONE OVER

        LD      A,H             ;WITHIN SECTOR SO
        SUB     8               ;RE-ADJUST
        LD      H,A
        RET


TEST:
        CALL    SLEFT           ;SCROLL LEFT
        CALL    SRIGHT          ;SCROLL RIGHT
        RET



        END
```

Hexadecimal Listing

```
7D00   C3  46  7D  21  1F  40  0E  10
7D08   E5  06  20  CB  16  2B  10  FB
7D10   E1  3E  00  8F  B6  77  CD  37
7D18   7D  0D  20  EC  C9  21  00  48
7D20   0E  10  E5  06  20  CB  1E  23
7D28   10  FB  E1  3E  00  1F  B6  77
7D30   CD  37  7D  0D  20  EC  C9  24
7D38   7C  E6  07  C0  7D  C6  20  6F

7D40   D8  7C  D6  08  67  C9  CD  03
7D48   7D  CD  1D  7D  C9  00  00  00
7D50   00  00  00  00  00  00  00  00
7D58   00  00  00  00  00  00  00  00
7D60   00  00  00  00  00  00  00  00
7D68   00  00  00  00  00  00  00  00
7D70   00  00  00  00  00  00  00  00
7D78   00  00  00  00  00  00  00  00
```

AUTO LINE NUMBER

This routine produces line numbers automatically when the user is typing in a BASIC program. Like the Clock and Trace programs given

in Chapter 11, this uses interrupts. The routine LOADS the ASCII line number into the system variable LAST–K every 1/50th of a second. This causes the line number to be placed in the edit area and lower screen of the Spectrum. To enable the auto line facility, key in the instruction RAND USR 32333. To turn off the auto line first DELETE the line number currently being edited and then enter RAND USR 32330.

Don't forget to CLEAR memory to keep the machine code safe. CLEAR 32329 is suitable for this.

Assembler Listing

```
        ORG     32330D


ECHOE   EQU     23682           ;COL NUMBER AND ROW NUMBER
LASTK   EQU     23560           ;LAST KEY TO BE PRESSED
FLAGS   EQU     23611           ;KEYBOARD FLAGS
EPCC    EQU     23625           ;CURRENT LINE NUMBER
PPC     EQU     23621           ;CURRENT LINE EXECUTING



DISINT:
        IM      1               ;ENABLE DEFAULT INTERRUPTS
        RET

ENABLE:
        XOR     A               ;SET A TO ZERO
        LD      (STATE),A       ;NOT OUTPUTTING ASCII CHARS
        LD      A,28H           ;SET I REG TO PAGE 28H
        LD      I,A
        IM      2               ;AND ENABLE INTERRUPT MODE 2
        EI
        RET

        ORG     7E5CH           ;START OF INTERRUPT ROUTINE

        CALL    38H             ;SCAN KEYBOARD FIRST
        DI                      ;DISABLE INTERRUPTS FIRST
        PUSH AF                 ;SAVE REGISTERS
        PUSH BC
        PUSH DE
        PUSH HL
        PUSH IX
```

```
        LD      A,(STATE)       ;GET STATE
        AND     A               ;TEST FOR ZERO
        JR      NZ,DML          ;ALL READY DOING A LINE


        LD      A,(PPC+1)       ;ARE WE EXECUTING A LINE?
        CP      255
        JR      NZ,BYE          ;YES SO EXIT FROM ROUTINE


        LD      A,(ECHOE)
        CP      20H
        JR      NZ,BYE

        LD      A,(ECHOE+1)
        CP      17H
        JR      NZ,BYE          ;ARE WE AT BOTTOM OF SCREEN
;ARRIVE HERE IF WE ARE AT THE BOTTOM OF THE SCREEN


        LD      A,(LASTK)       ;GET LASTK
        CP      0DH             ;IF NOT RETURN
        JR      NZ,BYE          ;THEN EXIT FROM ROUTINE

FIRST:  LD      A,4             ;SET UP NO OF CHARS
                                ;TO PRINT
        LD      (STATE),A
        LD      HL,DECTL        ;START OF TABLE
        LD      (CDATA),HL


        LD      HL,(EPCC)       ;GET CURRENT LINE NO
        LD      DE,000AH        ;GET STEP NUMBER
        ADD     HL,DE           ;GET NEXT LINE NUMBER
        LD      (LINE),HL       ;AND SAVE

DML:    LD      A,(STATE)       ;GET STATE
        DEC     A               ;ONE LESS CHAR
        LD      (STATE),A       ;TO PRINT
        LD      HL,(LINE)       ;GET LINE NUMBER
        CALL    CONV            ;OUTPUT ASCII TO LASTK
```

```
BYE:                                    ;RESTORE REGS
        POP IX
        POP HL
        POP DE
        POP BC
        POP AF
        EI
        RET

DECTL:  DEFW     1000D                  ;START OF TABLE
DECTX:  DEFW     100D
        DEFW     10D
        DEFW     1D

CONV:   LD       IX,(CDATA)             ;GET CURRENT TABLE POINTER
NDIGIT: LD       C,(IX+0)               ;GET LOW BYTE OF MULTIPLES
                                        ;OF TENS
        LD       B,(IX+1)               ;GET HIGH BYTE OF MULTIPLE
        LD       A,'0'-1                ;SET UP A REG WITH 30 HEX
        AND      A                      ;RESET CARRY
FIDIG:  INC      A                      ;ADD 1 TO A REGISTER
        SBC      HL,BC                  ;UNTIL WE GET A CARRY
        JR       NC,FIDIG
        ADD      HL,BC                  ;CORRECT NUMBER IN HL
        LD       (LINE),HL              ;AND SAVE IT!
        CALL     OUTP2                  ;OUTPUT ASCII CHAR IN A
                                        ;REGISTER
        INC      IX                     ;POINT TO NEXT MULTIPLE
        INC      IX
        PUSH     IX                     ;TRANSFER IX TO HL
                                        ;REGISTER PAIR
        POP      HL
        LD       (CDATA),HL             ;AND SAVE
        RET


OUTP2:
        LD       (LASTK),A              ;PLACE CHAR IN LASTK
        LD       HL,FLAGS               ;SIGNIFY WE ....
        SET      5,(HL)                 ;PRESSED A KEY
        RET
```

```
CDATA:  DEFW    0               ;CURRENT LINE DATA
STATE:  DB      0               ;NO OF CHARS TO PRINT
LINE:   DEFW    0               ;LINE NUMBER

        END
```

Hexadecimal Listing

```
7E4A  ED  56  C9  AF  32  E9  7E  3E
7E52  28  ED  47  ED  5E  FB  C9  00
7E5A  00  00  CD  38  00  F3  F5  C5
7E62  D5  E5  DD  E5  3A  E9  7E  A7
7E6A  20  31  3A  46  5C  FE  FF  20
7E72  37  3A  82  5C  FE  20  20  30
7E7A  3A  83  5C  FE  17  20  29  3A
7E82  08  5C  FE  0D  20  22  3E  04

7E8A  32  E9  7E  21  B2  7E  22  E7
7E92  7E  2A  49  5C  11  0A  00  19
7E9A  22  EA  7E  3A  E9  7E  3D  32
7EA2  E9  7E  2A  EA  7E  CD  BA  7E
7EAA  DD  E1  E1  D1  C1  F1  FB  C9
7EB2  E8  03  64  00  0A  00  01  00
7EBA  DD  2A  E7  7E  DD  4E  00  DD
7EC2  46  01  3E  2F  A7  3C  ED  42

7ECA  30  FB  09  22  EA  7E  CD  DE
7ED2  7E  DD  23  DD  23  DD  E5  E1
7EDA  22  E7  7E  C9  32  08  5C  21
7EE2  3B  5C  CB  EE  C9  00  00  00
7EEA  00  00  3A  46  5C  FE  FF  20
7EF2  37  3A  82  5C  FE  20  20  30
7EFA  3A  83  5C  FE  17  20
```

SORT

Another program which can be used with BASIC, this sort routine which allows you to sort strings into alphabetical order. The routine, when called in BASIC searches for the dimensional array A$. It should be first set up with the number of objects to sort and the length of each string. If the string is not found or the length is too large then it will exit from the sort routine with an appropriate error message. When you wish to sort the string you simply call the machine code from BASIC by using the instruction RAND USR 32000. This will then sort

out the string in ascending order. The method used to sort out the strings is known as a 'Bubble Sort'. This method of sorting is not the most efficient. However, under one second to sort out 100 strings of 25 characters in length is not slow!

The BASIC listing below demonstrates how the machine code program is used:

```
   5 LET sort=32000
  10 DIM a$(100,25)
  20 FOR p=1 to 100
  30 FOR c=1 to 25
  40 LET a$(p,c)=CHR$ ((RND*26)+
65)
  50 NEXT c
  60 NEXT p
  70 PRINT #0;"Press L to list,S
 to sort"
  80 LET k$=INKEY$: IF k$="" THE
N GO TO 80
  90 IF k$="L" OR k$="l" THEN GO
 SUB 120: GO TO 70
 100 IF k$<>"s" AND k$<>"S" THEN
 GO TO 80
 110 CLS: PRINT "sorting":RANDOM
IZE USR sort: BEEP 1,1: GO SUB 1
20: STOP
 120 FOR p=1 TO 100
 130 PRINT a$(p)
 140 NEXT p
 150 RETURN
```

Here are the listings for the sort routine:

Assembler Listing

```
        ORG     32000D
VARS    EQU     23627D

START:
        LD      HL,(VARS)       ;SET HL TO POINT TO
                                ;VARIABLE AREA
TEST:   LD      A,(HL)          ;GET 1ST BYTE OF VARIABLE
        CP      128             ;END OF VARS MARKER?
        JR      Z,NOTFOUND      ;FINISHED LOOKING AT VARS
```

```
        CP      193             ;IS IT A$?
        JP      Z,FOUND         ;YES FOUND IT!


        AND     11100000B       ;MASK OFF TOP THREE BITS
        CP      01100000B       ;SINGLE DATA
        JR      Z,ADISIX        ;YES ADD 6
        CP      11100000B       ;'FOR NEXT' VARIABLE?
        JR      Z,ADI19         ;YES ADD 19
        CP      10100000B       ;VARIABLE NAME LARGER
                                ;THEN ONE LETTER?
        JR      Z,SKIPC         ;YES THEN SKIP PASS
                                ;VARIABLE NAME

;010 OR  110
        INC     HL
        LD      E,(HL)          ;GET LENGTH LOW
        INC     HL
        LD      D,(HL)          ;GET LENGTH HIGH
        INC     HL
        ADD     HL,DE           ;SKIP PASS VARIABLE
        JP      TEST            ;TEST FOR NEXT VARIABLE




NOTFOUND:
        RST     08
        DB      01              ;VARIABLE NOT FOUND ERROR!
ERROR:
        RST     08              ;SUBSCRIPT WRONG ERROR!
        DB      02
ADI19:
        LD      DE,19           ;GO PASS VARIABLE
        ADD     HL,DE
        JP      TEST            ;TEST NEXT VARIABLE

SKIPC:
        INC     HL              ;SKIP PASS VARIABLE.
        BIT     7,(HL)          ;NAME
        JR      Z,SKIPC         ;TILL BIT 7 IS SET

ADISIX:
        LD      DE,6            ;GO PASS VARIABLE
```

```
        ADD     HL,DE
        JP      TEST            ;TEST NEXT VARIABLE

FOUND:
        INC     HL
        INC     HL
        INC     HL              ;POINT TO NUMBER OF DIMS
                                ;MUST BE TWO OR LESS


        LD      A,(HL)
        CP      2
        JR      NZ,ERROR        ;SHOULD BE TWO DIMENSIONS.

        INC     HL              ;POINT TO NUMBER OF
                                ;ELEMENTS
        LD      B,(HL)          ;NUMBER OF ELEMENTS
        INC     HL              ;GET HIGH BYTE!
        LD      A,(HL)
        AND     A
        JR      NZ,ERROR        ;LARGER THEN 255 ELEMENTS
        INC     HL
        LD      C,(HL)          ;LENGTH OF STRINGS
        INC     HL
        LD      A,(HL)
        AND     A
        JR      NZ,ERROR        ;LARGER THEN 255 CHARACTERS
        INC     HL
        ;HL NOW POINTS TO START OF STRING

        LD      A,C             ;SAVE SIZE
        LD      (SIZE),A


SORT:

;HL POINTS TO START OF STRING
;B CONTAINS NUMBER OF STRINGS
;C CONTAINS LENGTH OF STRING
```

```
NEXTS:  PUSH    BC              ;SAVE NUMBER AND LENGTH
        XOR     A               ;RESET SWAP FLAG
        LD      (FLAG),A
        PUSH    HL              ;SAVE ADDRESS OF FIRST
                                ;STRING
NEXTEL:
        PUSH    HL              ;SAVE ADDRESS OF STRING
        LD      E,C             ;GET LENGTH OF STRING
        LD      D,0             ;AND PLACE IN DE REGISTER
        ADD     HL,DE           ;POINT TO SECOND STRING
        EX      DE,HL           ;AND PLACE IN
                                ;THE DE REGISTER
        POP     HL              ;RESTORE ADDRESS OF STRING
        CALL    COMPARE         ;COMPARE THE TWO STRINGS
        CALL    C,SWAP          ;IN ASCENDING ORDER

        EX      DE,HL           ;HL NOW POINTS TO
                                ;NEXT STRING

        DJNZ    NEXTEL          ;REPEAT COMPARISION UNTIL
                                ;DONE UP TO CURRENT NUMBER
                                ;OF STRINGS
        POP     HL              ;GET ADDRESS OF
                                ;FIRST STRING
        POP     BC              ;RESTORE COUNTERS
        LD      A,(FLAG)        ;GET SWAP FLAG
        AND     A               ;TEST FOR ZERO
        RET     Z               ;NO SWAPS MADE SO SORTED
        DEC     B               ;ONE LESS TO SORT
        JR      NZ,NEXTS
        RET


COMPARE:
        PUSH    HL              ;SAVE REGISTERS
        PUSH    DE
        PUSH    BC
COMPARS:                        ;COMPARE STRINGS
                                ;ONE POINTED BY THE HL PAIR
                                ;AND ONE POINTED BY
                                ;THE DE PAIR
```

```
        LD      A,(DE)          ;GET CHARACTER
        SUB     (HL)            ;COMPARE AGAINST THE
                                ;SAME ONE IN
                                ;THE SECOND STRING
        JR      NZ,BYEFC        ;NOT EQUAL EXIT FROM
                                ;COMPARISON
        INC     HL              ;POINT TO NEXT CHARACTER
        INC     DE              ;POINT TO NEXT CHARACTER
        DEC     C               ;REPEAT UNTIL COMPARED
                                ;ALL CHARACTERS
        JR      NZ,COMPARS

BYEFC:  POP     BC              ;RESTORE REGISTERS
        POP     DE
        POP     HL
        RET


SWAP:
        ;SWAP THE TWO STRINGS POINTED
        ;BY THE HL PAIR AND THE DE PAIR

        PUSH    BC              ;SAVE REGISTERS
        PUSH    DE
        PUSH    HL

        LD      A,(SIZE)        ;GET SIZE
        LD      C,A             ;PLACE IN THE LOW BYTE
                                ;OF THE COUNTER
        LD      B,0             ;NOT LARGER THEN 255

        LD      DE,BUFF         ;DE POINTS TO THE BUFFER
        LDIR                    ;MOVE THE STRING FROM HL
                                ;TO THE BUFFER

        POP     DE              ;PUT ORIGINAL HL IN DE
        PUSH    DE

        LD      C,A             ;GET COUNTER
        LDIR                    ;MOVE TO SECOND STRING

        LD      HL,BUFF         ;POINT TO BUFFER
        LD      C,A             ;GET COUNT
        LDIR                    ;AND SWAP
```

```
        LD      (FLAG),A        ;SIGNIFY A SWAP WAS MADE

        POP     HL              ;RESTORE REGISTERS
        POP     DE
        POP     BC
        RET

BUFF:   DS      255             ;BUFFER
FLAG:   DB      0               ;SWAP FLAG
SIZE:   DB      0               ;SIZE OF STRING
        END
```

Hexadecimal Listing

```
7D00   2A  4B  5C  7E  FE  80  28  1C
7D08   FE  C1  CA  3B  7D  E6  E0  FE
7D10   60  28  21  FE  E0  28  11  FE
7D18   A0  28  14  23  5E  23  56  23
7D20   19  C3  03  7D  CF  01  CF  02
7D28   11  13  00  19  C3  03  7D  23
7D30   CB  7E  28  FB  11  06  00  19
7D38   C3  03  7D  23  23  23  7E  FE

7D40   02  20  E3  23  46  23  7E  A7
7D48   20  DC  23  4E  23  7E  A7  20
7D50   D5  23  79  32  A7  7E  C5  AF
7D58   32  A6  7E  E5  E5  59  16  00
7D60   19  EB  E1  CD  77  7D  DC  87
7D68   7D  EB  10  F0  E1  C1  3A  A6
7D70   7E  A7  C8  05  20  E0  C9  E5
7D78   D5  C5  1A  96  20  05  23  13

7D80   0D  20  F7  C1  D1  E1  C9  C5
7D88   D5  E5  3A  A7  7E  4F  06  00
7D90   11  A7  7D  ED  B0  D1  D5  4F
7D98   ED  B0  21  A7  7D  4F  ED  B0
7DA0   32  A6  7E  E1  D1  C1  C9  00
7DA8   00  00  00  00  00  00  00  00
7DB0   00  00  00  00  00  00  00  00
7DB8   00  00  00  00  00  00  00  00
```

RECURSION

This program is similar to the music routine given in Chapter 9 but is slightly more elaborate and complex. The tune I have given is the one I translated (from the Spectrum manual) from the section on the BEEP command. You can however write your own music. See the table given in Chapter nine. The routine is called by setting the IX register to point to the music data. The data represents the notes to be played and the duration. Each note and duration is represented by two bytes making a total of four. The first two bytes make up the frequency of the note and the second two the duration. The nice thing about this music routine is that it has the ability to play substrings of music. The routine scans first of all for the frequency in the table. If the low byte of the frequency is a one then this indicates that the following two bytes are the address of a substring to be played. The end of a string of music is indicated by having the byte 0. Substrings can be nested to many levels dependent on the RAM you have left. The whole principle behind this routine is that of recursion. It's a routine which calls itself, in the same way as BASIC subroutines can.

Assembler Listing

```
        ORG     32000D

BEEPER  EQU     03B5H           ;ADDRESS OF BEEPER ROUTINE
        LD      IX,FRERE        ;POINT TO MUSIC
        CALL    PLAY            ;AND PLAY IT SAM!
        RET
PLAY:
        PUSH    IX              ;SAVE STRING POSTION
        LD      L,(IX+0)        ;LOW PITCH
        LD      H,(IX+1)        ;HIGH PITCH
        LD      E,(IX+2)        ;LOW DURATION
        LD      D,(IX+3)        ;HIGH DURATION
        LD      A,L             ;LOOK AT LOW PITCH
        CP      01
        JR      Z,PLS           ;PLAY SUBSTRING
        JR      C,BYE           ;ZERO SO BYE
        CALL    BEEPER          ;PLAY NOTE
        POP     IX              ;GET STRING POSTION
        LD      DE,4            ;NEXT NOTE AND DURATION
        ADD     IX,DE
        JR      PLAY            ;KEEP PLAYING SAM!
```

```
PLS:
        INC     IX              ;POINT
        INC     IX              ;TO RETURN POSTION
        INC     IX
        LD      L,H
    ;ADJUST SUBSTRING ADDRESS
        LD      H,E
        POP     AF              ;GET RID OF OLD STRING
                                ;ADDRESS
        PUSH    IX              ;PUT IN NEW STRING ADDRESS

        PUSH    HL              ;TRANFER SUBSTRING
        POP     IX              ;ADDRESS TO IX REGISTER
        CALL    PLAY            ;PLAY SUBSTRING
        POP     IX              ;RETURNED FROM PLAYING
                                ;SUBSTRING
        JR      PLAY            ;KEEP PLAYING
BYE:
        POP     IX
        RET




FRERE:
        DB      01
        DEFW    FRERE1
        DB      01
        DEFW    FRERE1

FRERE1: DB      01
        DEFW    TUNE1
        DB      01
        DEFW    TUNE1



        DB      01
        DEFW    TUNE2
        DB      01
        DEFW    TUNE2
```

```
        DB      01
        DEFW    TUNE3
        DB      01
        DEFW    TUNE3



        DB      01
        DEFW    TUNE4
        DB      01
        DEFW    TUNE4


TUNE1:
        DEFW    66AH
        DEFW    105H
        DEFW    5B3H
        DEFW    125H
        DEFW    560H
        DEFW    9BH
        DEFW    5B3H
        DEFW    92H
        DEFW    66AH
        DEFW    105H
        DEFW    00


TUNE2:
        DEFW    560H
        DEFW    137H
        DEFW    4C6H
        DEFW    15DH
        DEFW    43DH
        DEFW    188H
        DEFW    00
```

```
TUNE3:
        DEFW    43DH
        DEFW    126H
        DEFW    3FFH
        DEFW    67H
        DEFW    43DH
        DEFW    0C4H
        DEFW    4C6H
        DEFW    0AEH
        DEFW    560H
        DEFW    9BH
        DEFW    5B3H
        DEFW    92H
        DEFW    66AH
        DEFW    105H
        DEFW    00


TUNE4:
        DEFW    66AH
        DEFW    105H
        DEFW    89AH
        DEFW    0C4H
        DEFW    66AH
        DEFW    20AH
        DEFW    00
        END
```

Hexadecimal Listing

```
7D00   DD  21  41  7D  CD  08  7D  C9
7D08   DD  E5  DD  6E  00  DD  66  01
7D10   DD  5E  02  DD  56  03  7D  FE
7D18   01  28  0E  38  21  CD  B5  03
7D20   DD  E1  11  04  00  DD  19  18
7D28   DF  DD  23  DD  23  DD  23  6C
7D30   63  F1  DD  E5  E5  DD  E1  CD
7D38   08  7D  DD  E1  18  CA  DD  E1
```

```
7D40   C9  01  47  7D  01  47  7D  01
7D48   5F  7D  01  5F  7D  01  75  7D
7D50   01  75  7D  01  83  7D  01  83
7D58   7D  01  A1  7D  01  A1  7D  6A
7D60   06  05  01  B3  05  25  01  60
7D68   05  9B  00  B3  05  92  00  6A
7D70   06  05  01  00  00  60  05  37
7D78   01  C6  04  5D  01  3D  04  88

7D80   01  00  00  3D  04  26  01  FF
7D88   03  67  00  3D  04  C4  00  C6
7D90   04  AE  00  60  05  9B  00  B3
7D98   05  92  00  6A  06  05  01  00
7DA0   00  6A  06  05  01  9A  08  C4
7DA8   00  6A  06  0A  02  00  00  6C
7DB0   63  F1  DD  E5  E5  DD  E1  CD
7DB8   08  7D  DD  E1  18  CA  DD  E1

7DC0   C9  01  47  7D  01  47  7D  01
7DC8   5F  7D  01  5F  7D  01  75  7D
7DD0   01  75  7D  01  83  7D  01  83
7DD8   7D  01  A1  7D  01  A1  7D  6A
7DE0   06  05  01  B3  05  25  01  60
7DE8   05  9B  00  B3  05  92  00  6A
7DF0   06  05  01  00  00  60  05  37
7DF8   01  C6  04  5D  01  3D  04  88
```

Appendix 1
Z80 instructions listed by mnemonic

8E	142	ADC A,(HL)
DD 8E dd	221 142 dd	ADC A,(IX'd)
FD 8E dd	253 142 dd	ADC A,(IY'd)
8F	143	ADC A,A
88	136	ADC A,B
89	137	ADC A,C
8A	138	ADC A,D
8B	139	ADC A,E
8C	140	ADC A,H
8D	141	ADC A,L
CE XX	206 XX	ADC A,N
ED 4A	237 74	ADC HL,BC
ED 5A	237 90	ADC HL,DE
ED 6A	237 106	ADC HL,HL
ED 7A	237 122	ADC HL,SP
86	134	ADD A,(HL)
DD 86 dd	221 134 dd	ADD A,(IX'd)
FD 86 dd	253 134 dd	ADD A,(IY'd)
87	135	ADD A,A
80	128	ADD A,B
81	129	ADD A,C
82	130	ADD A,D
83	131	ADD A,E
84	132	ADD A,H
85	133	ADD A,L
C6 XX	198 XX	ADD A,N
09	9	ADD HL,BC
19	25	ADD HL,DE
29	41	ADD HL,HL
39	57	ADD HL,SP
DD 09	221 9	ADD IX,BC
DD 19	221 25	ADD IX,DE
DD 29	221 41	ADD IX,IX
DD 39	221 57	ADD IX,SP
FD 09	253 9	ADD IY,BC

FD 19	253 25	ADD IY,DE
FD 29	253 41	ADD IY,IY
FD 39	253 57	ADD IY,SP
A6	166	AND (HL)
DD A6 dd	221 166 dd	AND (IX'd)
FD A6 dd	253 166 dd	AND (IY'd)
A7	167	AND A
A0	160	AND B
A1	161	AND C
A2	162	AND D
A3	163	AND E
A4	164	AND H
A5	165	AND L
E6 XX	230 XX	AND N
CB 46	203 70	BIT 0,(HL)
DD CB dd 46	221 203 dd 70	BIT 0,(IX'd)
FD CB dd 46	253 203 dd 70	BIT 0,(IY'd)
CB 47	203 71	BIT 0,A
CB 40	203 64	BIT 0,B
CB 41	203 65	BIT 0,C
CB 42	203 66	BIT 0,D
CB 43	203 67	BIT 0,E
CB 44	203 68	BIT 0,H
CB 45	203 69	BIT 0,L
CB 4E	203 78	BIT 1,(HL)
DD CB dd 4E	221 203 dd 78	BIT 1,(IX'd)
FD CB dd 4E	253 203 dd 78	BIT 1,(IY'd)
CB 4F	203 79	BIT 1,A
CB 48	203 72	BIT 1,B
CB 49	203 73	BIT 1,C
CB 4A	203 74	BIT 1,D
CB 4B	203 75	BIT 1,E
CB 4C	203 76	BIT 1,H
CB 4D	203 77	BIT 1,L
CB 56	203 86	BIT 2,(HL)
DD CB dd 56	221 203 dd 86	BIT 2,(IX'd)
FD CB dd 56	253 203 dd 86	BIT 2,(IY'd)
CB 57	203 87	BIT 2,A
CB 50	203 80	BIT 2,B
CB 51	203 81	BIT 2,C
CB 52	203 82	BIT 2,D
CB 53	203 83	BIT 2,E
CB 54	203 84	BIT 2,H
CB 55	203 85	BIT 2,L

CB 5E	203 94	BIT 3,(HL)
DD CB dd 5E	221 203 dd 94	BIT 3,(IX'd)
FD CB dd 5E	253 203 dd 94	BIT 3,(IY'd)
CB 5F	203 95	BIT 3,A
CB 58	203 88	BIT 3,B
CB 59	203 89	BIT 3,C
CB 5A	203 90	BIT 3,D
CB 5B	203 91	BIT 3,E
CB 5C	203 92	BIT 3,H
CB 5D	203 93	BIT 3,L
CB 66	203 102	BIT 4,(HL)
DD CB dd 66	221 203 dd 102	BIT 4,(IX'd)
FD CB dd 66	253 203 dd 102	BIT 4,(IY'd)
CB 67	203 103	BIT 4,A
CB 60	203 96	BIT 4,B
CB 61	203 97	BIT 4,C
CB 62	203 98	BIT 4,D
CB 63	203 99	BIT 4,E
CB 64	203 100	BIT 4,H
CB 65	203 101	BIT 4,L
CB 6E	203 110	BIT 5,(HL)
DD CB dd 6E	221 203 dd 110	BIT 5,(IX'd)
FD CB dd 6E	253 203 dd 110	BIT 5,(IY'd)
CB 6F	203 111	BIT 5,A
CB 68	203 104	BIT 5,B
CB 69	203 105	BIT 5,C
CB 6A	203 106	BIT 5,D
CB 6B	203 107	BIT 5,E
CB 6C	203 108	BIT 5,H
CB 6D	203 109	BIT 5,L
CB 76	203 118	BIT 6,(HL)
DD CB dd 76	221 203 dd 118	BIT 6,(IX'd)
FD CB dd 76	253 203 dd 118	BIT 6,(IY'd)
CB 77	203 119	BIT 6,A
CB 70	203 112	BIT 6,B
CB 71	203 113	BIT 6,C
CB 72	203 114	BIT 6,D
CB 73	203 115	BIT 6,E
CB 74	203 116	BIT 6,H
CB 75	203 117	BIT 6,L
CB 7E	203 126	BIT 7,(HL)
DD CB dd 7E	221 203 dd 126	BIT 7,(IX'd)
FD CB dd 7E	253 203 dd 126	BIT 7,(IY'd)
CB 7F	203 127	BIT 7,A

CB 78	203 120	BIT 7,B
CB 79	203 121	BIT 7,C
CB 7A	203 122	BIT 7,D
CB 7B	203 123	BIT 7,E
CB 7C	203 124	BIT 7,H
CB 7D	203 125	BIT 7,L
DC XXXX	220 XXXX	CALL C,NN
FC XXXX	252 XXXX	CALL M,NN
D4 XXXX	212 XXXX	CALL NC,NN
CD XXXX	205 XXXX	CALL NN
C4 XXXX	196 XXXX	CALL NZ,NN
F4 XXXX	244 XXXX	CALL P,NN
EC XXXX	236 XXXX	CALL PE,NN
E4 XXXX	228 XXXX	CALL PO,NN
CC XXXX	204 XXXX	CALL Z,NN
3F	63	CCF
BE	190	CP (HL)
DD BE dd	221 190 dd	CP (IX'd)
FD BE dd	253 190 dd	CP (IY'd)
BF	191	CP A
B8	184	CP B
B9	185	CP C
BA	186	CP D
BB	187	CP E
BC	188	CP H
BD	189	CP L
FE XX	254 XX	CP N
ED A9	237 169	CPD
ED B9	237 185	CPDR
ED A1	237 161	CPI
ED B1	237 177	CPIR
2F	47	CPL
27	39	DAA
35	53	DEC (HL)
DD 35 dd	221 53 dd	DEC (IX'd)
FD 35 dd	253 53 dd	DEC (IY'd)
3D	61	DEC A
05	5	DEC B
0B	11	DEC BC
0D	13	DEC C
15	21	DEC D
1B	27	DEC DE
1D	29	DEC E
25	37	DEC H

2B	43	DEC HL
DD 2B	221 43	DEC IX
FD 2B	253 43	DEC IY
2D	45	DEC L
3B	59	DEC SP
F3	243	DI
10	16 XX	DJNZ N
FB	251	EI
E3	227	EX (SP),HL
DD E3	221 227	EX (SP),IX
FD E3	253 227	EX (SP),IY
08	8	EX AF,AF'
EB	235	EX DE,HL
D9	217	EXX
76	118	HALT
ED 46	237 70	IM 0
ED 56	237 86	IM 1
ED 5E	237 94	IM 2
ED 78	237 120	IN A,(C)
DB XX	219 XX	IN A,(N)
ED 40	237 64	IN B,(C)
ED 48	237 72	IN C,(C)
ED 50	237 80	IN D,(C)
ED 58	237 88	IN E,(C)
ED 60	237 96	IN H,(C)
ED 68	237 104	IN L,(C)
34	52	INC (HL)
DD 34 dd	221 52 dd	INC (IX'd)
FD 34 dd	253 52 dd	INC (IY'd)
3C	60	INC A
04	4	INC B
03	3	INC BC
0C	12	INC C
14	20	INC D
13	19	INC DE
1C	28	INC E
24	36	INC H
23	35	INC HL
DD 23	221 35	INC IX
FD 23	253 35	INC IY
2C	44	INC L
33	51	INC SP
ED AA	237 170	IND
ED BA	237 186	INDR

ED A2	237 162	INI
ED B2	237 178	INIR
E9	233	JP (HL)
DD E9	221 233	JP (IX)
FD E9	253 233	JP (IY)
DA XXXX	218 XXXX	JP C,NN
FA XXXX	250 XXXX	JP M,NN
D2 XXXX	210	JP NC,NN
C3 XXXX	195 XXXX	JP NN
C2 XXXX	194 XXXX	JP NZ,NN
F2 XXXX	242 XXXX	JP P,NN
EA XXXX	234 XXXX	JP PE,NN
E2 XXXX	226 XXXX	JP PO,NN
CA XXXX	202 XXXX	JP Z,NN
38 XX	56 XX	JR C,N
18 XX	24 XX	JR N
30 XX	48 XX	JR NC,N
20 XX	32 XX	JR NZ,N
28 XX	40 XX	JR Z,N
0E XX	14 XX	LD C,N
02	2	LD (BC),A
12	18	LD (DE),A
77	119	LD (HL),A
70	112	LD (HL),B
71	113	LD (HL),C
72	114	LD (HL),D
73	115	LD (HL),E
74	116	LD (HL),H
75	117	LD (HL),L
36 XX	54 XX	LD (HL),N
DD 77 dd	221 119 dd	LD (IX'd),A
DD 70 dd	221 112 dd	LD (IX'd),B
DD 71 dd	221 113 dd	LD (IX'd),C
DD 72 dd	221 114 dd	LD (IX'd),D
DD 73 dd	221 115 dd	LD (IX'd),E
DD 74 dd	221 116 dd	LD (IX'd),H
DD 75 dd	221 117 dd	LD (IX'd),L
DD 36 dd XX	221 54 dd XX	LD (IX'd),N
FD 77 dd	253 119 dd	LD (IY'd),A
FD 70 dd	253 112 dd	LD (IY'd),B
FD 71 dd	253 113 dd	LD (IY'd),C
FD 72 dd	253 114 dd	LD (IY'd),D
FD 73 dd	253 115 dd	LD (IY'd),E
FD 74 dd	253 116 dd	LD (IY'd),H

FD 75 dd	253 117 dd	LD (IY'd),L
FD 36 dd XX	253 54 dd XX	LD (IY'd),N
32 XXXX	50 XXXX	LD (NN),A
ED 43 XXXX	237 67 XXXX	LD (NN),BC
ED 53 XXXX	237 83 XXXX	LD (NN),DE
22 XXXX	34 XXXX	LD (NN),HL
ED 63 XXXX	237 99 XXXX	LD (NN),HL
DD 22 XXXX	221 34 XXXX	LD (NN),IX
FD 22 XXXX	253 34 XXXX	LD (NN),IY
ED 73 XXXX	237 115 XXXX	LD (NN),SP
0A	10	LD A,(BC)
1A	26	LD A,(DE)
7E	126	LD A,(HL)
DD 7E dd	221 126 dd	LD A,(IX'd)
FD 7E dd	253 126 dd	LD A,(IY'd)
3A XXXX	58 XXXX	LD A,(NN)
7F	127	LD A,A
78	120	LD A,B
79	121	LD A,C
7A	122	LD A,D
7B	123	LD A,E
7C	124	LD A,H
ED 57	237 87	LD A,I
7D	125	LD A,L
3E XX	62 XX	LD A,N
ED 5F	237 95	LD A,R
46	70	LD B,(HL)
DD 46 dd	221 70 dd	LD B,(IX'd)
FD 46 dd	253 70 dd	LD B,(IY'd)
47	71	LD B,A
40	64	LD B,B
41	65	LD B,C
42	66	LD B,D
43	67	LD B,E
44	68	LD B,H
45	69	LD B,L
06 XX	6 XX	LD B,N
ED 4B XXXX	237 75 XXXX	LD BC,(NN)
01 XXXX	1 XXXX	LD BC,NN
4E	78	LD C,(HL)
DD 4E dd	221 78 dd	LD C,(IX'd)
FD 4E dd	253 78 dd	LD C,(IY'd)
4F	79	LD C,A
48	72	LD C,B

49	73	LD C,C
4A	74	LD C,D
4B	75	LD C,E
4C	76	LD C,H
4D	77	LD C,L
56	86	LD D,(HL)
DD 56 dd	221 86 dd	LD D,(IX'd)
FD 56 dd	253 86 dd	LD D,(IY'd)
57	87	LD D,A
50	80	LD D,B
51	81	LD D,C
52	82	LD D,D
53	83	LD D,E
54	84	LD D,H
55	85	LD D,L
16 XX	22 XX	LD D,N
ED 5B XXXX	237 91 XXXX	LD DE,(NN)
11	17 XXXX	LD DE,NN
5E	94	LD E,(HL)
DD 5E dd	221 94 dd	LD E,(IX'd)
FD 5E dd	253 94 dd	LD E,(IY'd)
5F	95	LD E,A
58	88	LD E,B
59	89	LD E,C
5A	90	LD E,D
5B	91	LD E,E
5C	92	LD E,H
5D	93	LD E,L
1E XX	30 XX	LD E,N
66	102	LD H,(HL)
DD 66 dd	221 102 dd	LD H,(IX'd)
FD 66 dd	253 102 dd	LD H,(IY'd)
67	103	LD H,A
60	96	LD H,B
61	97	LD H,C
62	98	LD H,D
63	99	LD H,E
64	100	LD H,H
65	101	LD H,L
26 XX	38 XX	LD H,N
2A XXXX	42 XXXX	LD HL,(NN)
ED 6B XXXX	237 107 XXXX	LD HL,(NN)
21 XXXX	33 XXXX	LD HL,NN
ED 47	237 71	LD I,A

DD 2A XXXX	221 42	LD IX,(NN)
DD 21 XXXX	221 33 XXXX	LD IX,NN
FD 2A XXXX	253 42	LD IY,(NN)
FD 21 XXXX	253 33 XXXX	LD IY,NN
6E	110	LD L,(HL)
DD 6E dd	221 110 dd	LD L,(IX'd)
FD 6E dd	253 110 dd	LD L,(IY'd)
6F	111	LD L,A
68	104	LD L,B
69	105	LD L,C
6A	106	LD L,D
6B	107	LD L,E
6C	108	LD L,H
6D	109	LD L,L
2E XX	46	LD L,N
ED 4F	237 79	LD R,A
ED 7B XXXX	237 123 XXXX	LD SP,(NN)
F9	249	LD SP,HL
DD F9	221 249	LD SP,IX
FD F9	253 249	LD SP,IY
31 XXXX	49 XXXX	LD SP,NN
ED A8	237 168	LDD
ED B8	237 184	LDDR
ED A0	237 160	LDI
ED B0	237 176	LDIR
ED 44	237 68	NEG
00	0	NOP
B6	182	OR (HL)
DD B6 dd	221 182 dd	OR (IX'd)
FD B6 dd	253 182 dd	OR (IY'd)
B7	183	OR A
B0	176	OR B
B1	177	OR C
B2	178	OR D
B3	179	OR E
B4	180	OR H
B5	181	OR L
F6 XX	246 XX	OR N
ED BB	237 187	OTDR
ED B3	237 179	OTIR
ED 79	237 121	OUT (C),A
ED 41	237 65	OUT (C),B
ED 49	237 73	OUT (C),C
ED 51	237 81	OUT (C),D

ED 59	237 89	OUT (C),E
ED 61	237 97	OUT (C),H
ED 69	237 105	OUT (C),L
D3 XX	211 XX	OUT (N),A
ED AB	237 171	OUTD
ED A3	237 163	OUTI
F1	241	POP AF
C1	193	POP BC
D1	209	POP DE
E1	225	POP HL
DD E1	221 225	POP IX
FD E1	253 225	POP IY
F5	245	PUSH AF
C5	197	PUSH BC
D5	213	PUSH DE
E5	229	PUSH HL
DD E5	221 229	PUSH IX
FD E5	253 229	PUSH IY
CB 86	203 134	RES 0,(HL)
DD CB dd 86	221 203 dd 134	RES 0,(IX'd)
FD CB dd 86	253 203 dd 134	RES 0,(IY'd)
CB 87	203 135	RES 0,A
CB 80	203 128	RES 0,B
CB 81	203 129	RES 0,C
CB 82	203 130	RES 0,D
CB 83	203 131	RES 0,E
CB 84	203 132	RES 0,H
CB 85	203 133	RES 0,L
CB 8E	203 142	RES 1,(HL)
DD CB dd 8E	221 203 dd 142	RES 1,(IX'd)
FD CB dd 8E	253 203 dd 142	RES 1,(IY'd)
CB 8F	203 143	RES 1,A
CB 88	203 136	RES 1,B
CB 89	203 137	RES 1,C
CB 8A	203 138	RES 1,D
CB 8B	203 139	RES 1,E
CB 8C	203 140	RES 1,H
CB 8D	203 141	RES 1,L
CB 96	203 150	RES 2,(HL)
DD CB dd 96	221 203 dd 150	RES 2,(IX'd)
FD CB dd 96	253 203 dd 150	RES 2,(IY'd)
CB 97	203 151	RES 2,A
CB 90	203 144	RES 2,B
CB 91	203 145	RES 2,C

CB 92	203 146	RES 2,D
CB 93	203 147	RES 2,E
CB 94	203 148	RES 2,H
CB 95	203 149	RES 2,L
CB 9E	203 158	RES 3,(HL)
DD CB dd 9E	221 203 dd 158	RES 3,(IX'd)
FD CB dd 9E	253 203 dd 158	RES 3,(IY'd)
CB 9F	203 159	RES 3,A
CB 98	203 152	RES 3,B
CB 99	203 153	RES 3,C
CB 9A	203 154	RES 3,D
CB 9B	203 155	RES 3,E
CB 9C	203 156	RES 3,H
CB 9D	203 157	RES 3,L
CB A6	203 166	RES 4,(HL)
DD CB dd A6	221 203 dd 166	RES 4,(IX'd)
FD CB dd A6	253 203 dd 166	RES 4,(IY'd)
CB A7	203 167	RES 4,A
CB A0	203 160	RES 4,B
CB A1	203 161	RES 4,C
CB A2	203 162	RES 4,D
CB A3	203 163	RES 4,E
CB A4	203 164	RES 4,H
CB A5	203 165	RES 4,L
CB AE	203 174	RES 5,(HL)
DD CB dd AE	221 203 dd 174	RES 5,(IX'd)
FD CB dd AE	253 203 dd 174	RES 5,(IY'd)
CB AF	203 175	RES 5,A
CB A8	203 168	RES 5,B
CB A9	203 169	RES 5,C
CB AA	203 170	RES 5,D
CB AB	203 171	RES 5,E
CB AC	203 172	RES 5,H
CB AD	203 173	RES 5,L
CB B6	203 182	RES 6,(HL)
DD CB dd B6	221 203 dd 182	RES 6,(IX'd)
FD CB dd B6	253 203 dd 182	RES 6,(IY'd)
CB B7	203 183	RES 6,A
CB B0	203 176	RES 6,B
CB B1	203 177	RES 6,C
CB B2	203 178	RES 6,D
CB B3	203 179	RES 6,E
CB B4	203 180	RES 6,H
CB B5	203 181	RES 6,L

CB BE	203 190	RES 7,(HL)
DD CB dd BE	221 203 dd 190	RES 7,(IX'd)
FD CB dd BE	253 203 dd 190	RES 7,(IY'd)
CB BF	203 191	RES 7,A
CB B8	203 184	RES 7,B
CB B9	203 185	RES 7,C
CB BA	203 186	RES 7,D
CB BB	203 187	RES 7,E
CB BC	203 188	RES 7,H
CB BD	203 189	RES 7,L
C9	201	RET
D8	216	RET C
F8	248	RET M
D0	208	RET NC
C0	192	RET NZ
F0	240	RET P
E8	232	RET PE
E0	224	RET PO
C8	200	RET Z
ED 4D	237 77	RETI
ED 45	237 69	RETN
CB 16	203 22	RL (HL)
DD CB dd 16	221 203 dd 20	RL (IX'd)
FD CB dd 16	253 203 dd 20	RL (IY'd)
CB 17	203 23	RL A
CB 10	203 16	RL B
CB 11	203 17	RL C
CB 12	203 18	RL D
CB 13	203 19	RL E
CB 14	203 20	RL H
CB 15	203 21	RL L
17	23	RLA
CB 06	203 6	RLC (HL)
DD CB dd 06	221 203 dd 6	RLC (IX'd)
FD CB dd 06	253 203 dd 6	RLC (IY'd)
CB 07	203 7	RLC A
CB 00	203 0	RLC B
CB 01	203 1	RLC C
CB 02	203 2	RLC D
CB 03	203 3	RLC E
CB 04	203 4	RLC H
CB 05	203 5	RLC L
07	7	RLCA
ED 6F	237 111	RLD

CB 1E	203 30	RR (HL)
DD CB dd 1E	221 203 dd 30	RR (IX'd)
FD CB dd 1E	253 203 dd 30	RR (IY'd)
CB 1F	203 31	RR A
CB 18	203 24	RR B
CB 19	203 25	RR C
CB 1A	203 26	RR D
CB 1B	203 27	RR E
CB 1C	203 28	RR H
CB 1D	203 29	RR L
1F	31	RRA
CB 0E	203 14	RRC (HL)
DD CB dd 0E	221 203 dd 14	RRC (IX'd)
FD CB dd 0E	253 203 dd 14	RRC (IY'd)
CB 0F	203 15	RRC A
CB 08	203 8	RRC B
CB 09	203 9	RRC C
CB 0A	203 10	RRC D
CB 0B	203 11	RRC E
CB 0C	203 12	RRC H
CB 0D	203 13	RRC L
0F	15	RRCA
ED 67	237 103	RRD
C7	199	RST 0
D7	215	RST 10
DF	223	RST 18
E7	231	RST 20
EF	239	RST 28
F7	247	RST 30
FF	255	RST 38
CF	207	RST 8
9E	158	SBC A,(HL)
DD 9E dd	221 158 dd	SBC A,(IX'd)
FD 9E dd	253 158 dd	SBC A,(IY'd)
9F	159	SBC A,A
98	152	SBC A,B
99	153	SBC A,C
9A	154	SBC A,D
9B	155	SBC A,E
9C	156	SBC A,H
9D	157	SBC A,L
DE XX	222 XX	SBC A,N
ED 42	237 66	SBC HL,BC
ED 52	237 82	SBC HL,DE

ED 62	237 98	SBC HL,HL
ED 72	237 114	SBC HL,SP
37	55	SCF
CB C6	203 198	SET 0,(HL)
DD CB dd C6	221 203 dd 198	SET 0,(IX'd)
FD CB dd C6	253 203 dd 198	SET 0,(IY'd)
CB C7	203 199	SET 0,A
CB C0	203 192	SET 0,B
CB C1	203 193	SET 0,C
CB C2	203 194	SET 0,D
CB C3	203 195	SET 0,E
CB C4	203 196	SET 0,H
CB C5	203 197	SET 0,L
CB CE	203 206	SET 1,(HL)
DD CB dd CE	221 203 dd 206	SET 1,(IX'd)
FD CB dd CE	253 203 dd 206	SET 1,(IY'd)
CB CF	203 207	SET 1,A
CB C8	203 200	SET 1,B
CB C9	203 201	SET 1,C
CB CA	203 202	SET 1,D
CB CB	203 203	SET 1,E
CB CC	203 204	SET 1,H
CB CD	203 205	SET 1,L
CB D6	203 214	SET 2,(HL)
DD CB dd D6	221 203 dd 214	SET 2,(IX'd)
FD CB dd D6	253 203 dd 214	SET 2,(IY'd)
CB D7	203 215	SET 2,A
CB D0	203 208	SET 2,B
CB D1	203 209	SET 2,C
CB D2	203 210	SET 2,D
CB D3	203 211	SET 2,E
CB D4	203 212	SET 2,H
CB D5	203 213	SET 2,L
CB DE	203 222	SET 3,(HL)
DD CB dd DE	221 203 dd 222	SET 3,(IX'd)
FD CB dd DE	253 203 dd 222	SET 3,(IY'd)
CB DF	203 223	SET 3,A
CB D8	203 216	SET 3,B
CB D9	203 217	SET 3,C
CB DA	203 218	SET 3,D
CB DB	203 219	SET 3,E
CB DC	203 220	SET 3,H
CB DD	203 221	SET 3,L
CB E6	203 230	SET 4,(HL)

DD CB dd E6	221 203 dd 230	SET 4,(IX'd)
FD CB dd E6	253 203 dd 230	SET 4,(IY'd)
CB E7	203 231	SET 4,A
CB E0	203 224	SET 4,B
CB E1	203 225	SET 4,C
CB E2	203 226	SET 4,D
CB E3	203 227	SET 4,E
CB E4	203 228	SET 4,H
CB E5	203 229	SET 4,L
CB EE	203 238	SET 5,(HL)
DD CB dd EE	221 203 dd 238	SET 5,(IX'd)
FD CB dd EE	253 203 dd 238	SET 5,(IY'd)
CB EF	203 239	SET 5,A
CB E8	203 232	SET 5,B
CB E9	203 233	SET 5,C
CB EA	203 234	SET 5,D
CB EB	203 235	SET 5,E
CB EC	203 236	SET 5,H
CB ED	203 237	SET 5,L
CB F6	203 246	SET 6,(HL)
DD CB dd F6	221 203 dd 246	SET 6,(IX'd)
FD CB dd F6	253 203 dd 246	SET 6,(IY'd)
CB F7	203 247	SET 6,A
CB F0	203 240	SET 6,B
CB F1	203 241	SET 6,C
CB F2	203 242	SET 6,D
CB F3	203 243	SET 6,E
CB F4	203 244	SET 6,H
CB F5	203 245	SET 6,L
CB FE	203 254	SET 7,(HL)
DD CB dd FE	221 203 dd 254	SET 7,(IX'd)
FD CB dd FE	253 203 dd 254	SET 7,(IY'd)
CB FF	203 255	SET 7,A
CB F8	203 248	SET 7,B
CB F9	203 249	SET 7,C
CB FA	203 250	SET 7,D
CB FB	203 251	SET 7,E
CB FC	203 252	SET 7,H
CB FD	203 253	SET 7,L
CB 26	203 38	SLA (HL)
DD CB dd 26	221 203 dd 38	SLA (IX'd)
FD CB dd 26	253 203 dd 38	SLA (IY'd)
CB 27	203 39	SLA A
CB 20	203 32	SLA B

CB 21	203 33	SLA C
CB 22	203 34	SLA D
CB 23	203 35	SLA E
CB 24	203 36	SLA H
CB 25	203 37	SLA L
CB 2E	203 46	SRA (HL)
DD CB dd 2E	221 203 dd 46	SRA (IX'd)
FD CB dd 2E	253 203 dd 46	SRA (IY'd)
CB 2F	203 47	SRA A
CB 28	203 40	SRA B
CB 29	203 41	SRA C
CB 2A	203 42	SRA D
CB 2B	203 43	SRA E
CB 2C	203 44	SRA H
CB 2D	203 45	SRA L
CB 3E	203 62	SRL (HL)
DD CB dd 3E	221 203 dd 62	SRL (IX'd)
FD CB dd 3E	253 203 dd 62	SRL (IY'd)
CB 3F	203 63	SRL A
CB 38	203 56	SRL B
CB 39	203 57	SRL C
CB 3A	203 58	SRL D
CB 3B	203 59	SRL E
CB 3C	203 60	SRL H
CB 3D	203 61	SRL L
96	150	SUB (HL)
DD 96 dd	221 150 dd	SUB (IX'd)
FD 96 dd	253 150 dd	SUB (IY'd)
97	151	SUB A
90	144	SUB B
91	145	SUB C
92	146	SUB D
93	147	SUB E
94	148	SUB H
95	149	SUB L
D6 XX	214 XX	SUB N
EE XX	238 XX	XOR N
AE	174	XOR (HL)
DD AE dd	221 174 dd	XOR (IX'd)
FD AE dd	253 174 dd	XOR (IY'd)
AF	175	XOR A
A8	168	XOR B
A9	169	XOR C

AA	170	XOR D
AB	171	XOR E
AC	172	XOR H
AD	173	XOR L

Appendix 2
Z8Ø instructions listed by opcode

00	0	NOP
01 XXXX	1 XXXX	LD BC,NN
02	2	LD (BC),A
03	3	INC BC
04	4	INC B
05	5	DEC B
06 XX	6 XX	LD B,N
07	7	RLCA
08	8	EX AF,AF'
09	9	ADD HL,BC
0A	10	LD A,(BC)
0B	11	DEC BC
0C	12	INC C
0D	13	DEC C
0E XX	14 XX	LD C,N
0F	15	RRCA
10	16 XX	DJNZ N
11	17 XXXX	LD DE,NN
12	18	LD (DE),A
13	19	INC DE
14	20	INC D
15	21	DEC D
16 XX	22 XX	LD D,N
17	23	RLA
18 XX	24 XX	JR N
19	25	ADD HL,DE
1A	26	LD A,(DE)
1B	27	DEC DE
1C	28	INC E
1D	29	DEC E
1E XX	30 XX	LD E,N
1F	31	RRA
20 XX	32 XX	JR NZ,N
21 XXXX	33 XXXX	LD HL,NN
22 XXXX	34 XXXX	LD (NN),HL

23	35	INC HL
24	36	INC H
25	37	DEC H
26 XX	38 XX	LD H,N
27	39	DAA
28 XX	40 XX	JR Z,N
29	41	ADD HL,HL
2A XXXX	42 XXXX	LD HL,(NN)
2B	43	DEC HL
2C	44	INC L
2D	45	DEC L
2E XX	46	LD L,N
2F	47	CPL
30 XX	48 XX	JR NC,N
31 XXXX	49 XXXX	LD SP,NN
32 XXXX	50 XXXX	LD (NN),A
33	51	INC SP
34	52	INC (HL)
35	53	DEC (HL)
36 XX	54 XX	LD (HL),N
37	55	SCF
38 XX	56 XX	JR C,N
39	57	ADD HL,SP
3A XXXX	58 XXXX	LD A,(NN)
3B	59	DEC SP
3C	60	INC A
3D	61	DEC A
3E XX	62 XX	LD A,N
3F	63	CCF
40	64	LD B,B
41	65	LD B,C
42	66	LD B,D
43	67	LD B,E
44	68	LD B,H
45	69	LD B,L
46	70	LD B,(HL)
47	71	LD B,A
48	72	LD C,B
49	73	LD C,C
4A	74	LD C,D
4B	75	LD C,E
4C	76	LD C,H
4D	77	LD C,L
4E	78	LD C,(HL)

4F	79	LD C,A
50	80	LD D,B
51	81	LD D,C
52	82	LD D,D
53	83	LD D,E
54	84	LD D,H
55	85	LD D,L
56	86	LD D,(HL)
57	87	LD D,A
58	88	LD E,B
59	89	LD E,C
5A	90	LD E,D
5B	91	LD E,E
5C	92	LD E,H
5D	93	LD E,L
5E	94	LD E,(HL)
5F	95	LD E,A
60	96	LD H,B
61	97	LD H,C
62	98	LD H,D
63	99	LD H,E
64	100	LD H,H
65	101	LD H,L
66	102	LD H,(HL)
67	103	LD H,A
68	104	LD L,B
69	105	LD L,C
6A	106	LD L,D
6B	107	LD L,E
6C	108	LD L,H
6D	109	LD L,L
6E	110	LD L,(HL)
6F	111	LD L,A
70	112	LD (HL),B
71	113	LD (HL),C
72	114	LD (HL),D
73	115	LD (HL),E
74	116	LD (HL),H
75	117	LD (HL),L
76	118	HALT
77	119	LD (HL),A
78	120	LD A,B
79	121	LD A,C
7A	122	LD A,D

7B	123	LD A,E
7C	124	LD A,H
7D	125	LD A,L
7E	126	LD A,(HL)
7F	127	LD A,A
80	128	ADD A,B
81	129	ADD A,C
82	130	ADD A,D
83	131	ADD A,E
84	132	ADD A,H
85	133	ADD A,L
86	134	ADD A,(HL)
87	135	ADD A,A
88	136	ADC A,B
89	137	ADC A,C
8A	138	ADC A,D
8B	139	ADC A,E
8C	140	ADC A,H
8D	141	ADC A,L
8E	142	ADC A,(HL)
8F	143	ADC A,A
90	144	SUB B
91	145	SUB C
92	146	SUB D
93	147	SUB E
94	148	SUB H
95	149	SUB L
96	150	SUB (HL)
97	151	SUB A
98	152	SBC A,B
99	153	SBC A,C
9A	154	SBC A,D
9B	155	SBC A,E
9C	156	SBC A,H
9D	157	SBC A,L
9E	158	SBC A,(HL)
9F	159	SBC A,A
A0	160	AND B
A1	161	AND C
A2	162	AND D
A3	163	AND E
A4	164	AND H
A5	165	AND L
A6	166	AND (HL)

A7	167	AND A
A8	168	XOR B
A9	169	XOR C
AA	170	XOR D
AB	171	XOR E
AC	172	XOR H
AD	173	XOR L
AE	174	XOR (HL)
AF	175	XOR A
B0	176	OR B
B1	177	OR C
B2	178	OR D
B3	179	OR E
B4	180	OR H
B5	181	OR L
B6	182	OR (HL)
B7	183	OR A
B8	184	CP B
B9	185	CP C
BA	186	CP D
BB	187	CP E
BC	188	CP H
BD	189	CP L
BE	190	CP (HL)
BF	191	CP A
C0	192	RET NZ
C1	193	POP BC
C2 XXXX	194 XXXX	JP NZ,NN
C3 XXXX	195 XXXX	JP NN
C4 XXXX	196 XXXX	CALL NZ,NN
C5	197	PUSH BC
C6 XX	198 XX	ADD A,N
C7	199	RST 0
C8	200	RET Z
C9	201	RET
CA XXXX	202 XXXX	JP Z,NN
CC XXXX	204 XXXX	CALL Z,NN
CD XXXX	205 XXXX	CALL NN
CE XX	206 XX	ADC A,N
CF	207	RST 8
D0	208	RET NC
D1	209	POP DE
D2 XXXX	210	JP NC,NN
D3 XX	211 XX	OUT (N),A

D4 XXXX	212 XXXX	CALL NC,NN
D5	213	PUSH DE
D6 XX	214 XX	SUB N
D7	215	RST 10
D8	216	RET C
D9	217	EXX
DA XXXX	218 XXXX	JP C,NN
DB XX	219 XX	IN A,(N)
DC XXXX	220 XXXX	CALL C,NN
DE XX	222 XX	SBC A,N
DF	223	RST 18
E0	224	RET PO
E1	225	POP HL
E2 XXXX	226 XXXX	JP PO,NN
E3	227	EX (SP),HL
E4 XXXX	228 XXXX	CALL PO,NN
E5	229	PUSH HL
E6 XX	230 XX	AND N
E7	231	RST 20
E8	232	RET PE
E9	233	JP (HL)
EA XXXX	234 XXXX	JP PE,NN
EB	235	EX DE,HL
EC XXXX	236 XXXX	CALL PE,NN
EE XX	238 XX	XOR N
EF	239	RST 28
F0	240	RET P
F1	241	POP AF
F2 XXXX	242 XXXX	JP P,NN
F3	243	DI
F4 XXXX	244 XXXX	CALL P,NN
F5	245	PUSH AF
F6 XX	246 XX	OR N
F7	247	RST 30
F8	248	RET M
F9	249	LD SP,HL
FA XXXX	250 XXXX	JP M,NN
FB	251	EI
FC XXXX	252 XXXX	CALL M,NN
FE XX	254 XX	CP N
FF	255	RST 38
CB 00	203 0	RLC B
CB 01	203 1	RLC C
CB 02	203 2	RLC D

CB 03	203 3	RLC E
CB 04	203 4	RLC H
CB 05	203 5	RLC L
CB 06	203 6	RLC (HL)
CB 07	203 7	RLC A
CB 08	203 8	RRC B
CB 09	203 9	RRC C
CB 0A	203 10	RRC D
CB 0B	203 11	RRC E
CB 0C	203 12	RRC H
CB 0D	203 13	RRC L
CB 0E	203 14	RRC (HL)
CB 0F	203 15	RRC A
CB 10	203 16	RL B
CB 11	203 17	RL C
CB 12	203 18	RL D
CB 13	203 19	RL E
CB 14	203 20	RL H
CB 15	203 21	RL L
CB 16	203 22	RL (HL)
CB 17	203 23	RL A
CB 18	203 24	RR B
CB 19	203 25	RR C
CB 1A	203 26	RR D
CB 1B	203 27	RR E
CB 1C	203 28	RR H
CB 1D	203 29	RR L
CB 1E	203 30	RR (HL)
CB 1F	203 31	RR A
CB 20	203 32	SLA B
CB 21	203 33	SLA C
CB 22	203 34	SLA D
CB 23	203 35	SLA E
CB 24	203 36	SLA H
CB 25	203 37	SLA L
CB 26	203 38	SLA (HL)
CB 27	203 39	SLA A
CB 28	203 40	SRA B
CB 29	203 41	SRA C
CB 2A	203 42	SRA D
CB 2B	203 43	SRA E
CB 2C	203 44	SRA H
CB 2D	203 45	SRA L
CB 2E	203 46	SRA (HL)

CB 2F	203 47	SRA A
CB 38	203 56	SRL B
CB 39	203 57	SRL C
CB 3A	203 58	SRL D
CB 3B	203 59	SRL E
CB 3C	203 60	SRL H
CB 3D	203 61	SRL L
CB 3E	203 62	SRL (HL)
CB 3F	203 63	SRL A
CB 40	203 64	BIT 0,B
CB 41	203 65	BIT 0,C
CB 42	203 66	BIT 0,D
CB 43	203 67	BIT 0,E
CB 44	203 68	BIT 0,H
CB 45	203 69	BIT 0,L
CB 46	203 70	BIT 0,(HL)
CB 47	203 71	BIT 0,A
CB 48	203 72	BIT 1,B
CB 49	203 73	BIT 1,C
CB 4A	203 74	BIT 1,D
CB 4B	203 75	BIT 1,E
CB 4C	203 76	BIT 1,H
CB 4D	203 77	BIT 1,L
CB 4E	203 78	BIT 1,(HL)
CB 4F	203 79	BIT 1,A
CB 50	203 80	BIT 2,B
CB 51	203 81	BIT 2,C
CB 52	203 82	BIT 2,D
CB 53	203 83	BIT 2,E
CB 54	203 84	BIT 2,H
CB 55	203 85	BIT 2,L
CB 56	203 86	BIT 2,(HL)
CB 57	203 87	BIT 2,A
CB 58	203 88	BIT 3,B
CB 59	203 89	BIT 3,C
CB 5A	203 90	BIT 3,D
CB 5B	203 91	BIT 3,E
CB 5C	203 92	BIT 3,H
CB 5D	203 93	BIT 3,L
CB 5E	203 94	BIT 3,(HL)
CB 5F	203 95	BIT 3,A
CB 60	203 96	BIT 4,B
CB 61	203 97	BIT 4,C
CB 62	203 98	BIT 4,D

CB 63	203 99	BIT 4,E
CB 64	203 100	BIT 4,H
CB 65	203 101	BIT 4,L
CB 66	203 102	BIT 4,(HL)
CB 67	203 103	BIT 4,A
CB 68	203 104	BIT 5,B
CB 69	203 105	BIT 5,C
CB 6A	203 106	BIT 5,D
CB 6B	203 107	BIT 5,E
CB 6C	203 108	BIT 5,H
CB 6D	203 109	BIT 5,L
CB 6E	203 110	BIT 5,(HL)
CB 6F	203 111	BIT 5,A
CB 70	203 112	BIT 6,B
CB 71	203 113	BIT 6,C
CB 72	203 114	BIT 6,D
CB 73	203 115	BIT 6,E
CB 74	203 116	BIT 6,H
CB 75	203 117	BIT 6,L
CB 76	203 118	BIT 6,(HL)
CB 77	203 119	BIT 6,A
CB 78	203 120	BIT 7,B
CB 79	203 121	BIT 7,C
CB 7A	203 122	BIT 7,D
CB 7B	203 123	BIT 7,E
CB 7C	203 124	BIT 7,H
CB 7D	203 125	BIT 7,L
CB 7E	203 126	BIT 7,(HL)
CB 7F	203 127	BIT 7,A
CB 80	203 128	RES 0,B
CB 81	203 129	RES 0,C
CB 82	203 130	RES 0,D
CB 83	203 131	RES 0,E
CB 84	203 132	RES 0,H
CB 85	203 133	RES 0,L
CB 86	203 134	RES 0,(HL)
CB 87	203 135	RES 0,A
CB 88	203 136	RES 1,B
CB 89	203 137	RES 1,C
CB 8A	203 138	RES 1,D
CB 8B	203 139	RES 1,E
CB 8C	203 140	RES 1,H
CB 8D	203 141	RES 1,L
CB 8E	203 142	RES 1,(HL)

CB 8F	203 143	RES 1,A
CB 90	203 144	RES 2,B
CB 91	203 145	RES 2,C
CB 92	203 146	RES 2,D
CB 93	203 147	RES 2,E
CB 94	203 148	RES 2,H
CB 95	203 149	RES 2,L
CB 96	203 150	RES 2,(HL)
CB 97	203 151	RES 2,A
CB 98	203 152	RES 3,B
CB 99	203 153	RES 3,C
CB 9A	203 154	RES 3,D
CB 9B	203 155	RES 3,E
CB 9C	203 156	RES 3,H
CB 9D	203 157	RES 3,L
CB 9E	203 158	RES 3,(HL)
CB 9F	203 159	RES 3,A
CB A0	203 160	RES 4,B
CB A1	203 161	RES 4,C
CB A2	203 162	RES 4,D
CB A3	203 163	RES 4,E
CB A4	203 164	RES 4,H
CB A5	203 165	RES 4,L
CB A6	203 166	RES 4,(HL)
CB A7	203 167	RES 4,A
CB A8	203 168	RES 5,B
CB A9	203 169	RES 5,C
CB AA	203 170	RES 5,D
CB AB	203 171	RES 5,E
CB AC	203 172	RES 5,H
CB AD	203 173	RES 5,L
CB AE	203 174	RES 5,(HL)
CB AF	203 175	RES 5,A
CB B0	203 176	RES 6,B
CB B1	203 177	RES 6,C
CB B2	203 178	RES 6,D
CB B3	203 179	RES 6,E
CB B4	203 180	RES 6,H
CB B5	203 181	RES 6,L
CB B6	203 182	RES 6,(HL)
CB B7	203 183	RES 6,A
CB B8	203 184	RES 7,B
CB B9	203 185	RES 7,C
CB BA	203 186	RES 7,D

CB BB	203 187	RES 7,E
CB BC	203 188	RES 7,H
CB BD	203 189	RES 7,L
CB BE	203 190	RES 7,(HL)
CB BF	203 191	RES 7,A
CB C0	203 192	SET 0,B
CB C1	203 193	SET 0,C
CB C2	203 194	SET 0,D
CB C3	203 195	SET 0,E
CB C4	203 196	SET 0,H
CB C5	203 197	SET 0,L
CB C6	203 198	SET 0,(HL)
CB C7	203 199	SET 0,A
CB C8	203 200	SET 1,B
CB C9	203 201	SET 1,C
CB CA	203 202	SET 1,D
CB CB	203 203	SET 1,E
CB CC	203 204	SET 1,H
CB CD	203 205	SET 1,L
CB CE	203 206	SET 1,(HL)
CB CF	203 207	SET 1,A
CB D0	203 208	SET 2,B
CB D1	203 209	SET 2,C
CB D2	203 210	SET 2,D
CB D3	203 211	SET 2,E
CB D4	203 212	SET 2,H
CB D5	203 213	SET 2,L
CB D6	203 214	SET 2,(HL)
CB D7	203 215	SET 2,A
CB D8	203 216	SET 3,B
CB D9	203 217	SET 3,C
CB DA	203 218	SET 3,D
CB DB	203 219	SET 3,E
CB DC	203 220	SET 3,H
CB DD	203 221	SET 3,L
CB DE	203 222	SET 3,(HL)
CB DF	203 223	SET 3,A
CB E0	203 224	SET 4,B
CB E1	203 225	SET 4,C
CB E2	203 226	SET 4,D
CB E3	203 227	SET 4,E
CB E4	203 228	SET 4,H
CB E5	203 229	SET 4,L
CB E6	203 230	SET 4,(HL)

CB E7	203 231	SET 4,A
CB E8	203 232	SET 5,B
CB E9	203 233	SET 5,C
CB EA	203 234	SET 5,D
CB EB	203 235	SET 5,E
CB EC	203 236	SET 5,H
CB ED	203 237	SET 5,L
CB EE	203 238	SET 5,(HL)
CB EF	203 239	SET 5,A
CB F0	203 240	SET 6,B
CB F1	203 241	SET 6,C
CB F2	203 242	SET 6,D
CB F3	203 243	SET 6,E
CB F4	203 244	SET 6,H
CB F5	203 245	SET 6,L
CB F6	203 246	SET 6,(HL)
CB F7	203 247	SET 6,A
CB F8	203 248	SET 7,B
CB F9	203 249	SET 7,C
CB FA	203 250	SET 7,D
CB FB	203 251	SET 7,E
CB FC	203 252	SET 7,H
CB FD	203 253	SET 7,L
CB FE	203 254	SET 7,(HL)
CB FF	203 255	SET 7,A
DD 09	221 9	ADD IX,BC
DD 19	221 25	ADD IX,DE
DD 21 XXXX	221 33 XXXX	LD IX,NN
DD 22 XXXX	221 34 XXXX	LD (NN),IX
DD 23	221 35	INC IX
DD 29	221 41	ADD IX,IX
DD 2A XXXX	221 42	LD IX,(NN)
DD 2B	221 43	DEC IX
DD 34 dd	221 52 dd	INC (IX'd)
DD 35 dd	221 53 dd	DEC (IX'd)
DD 36 dd XX	221 54 dd XX	LD (IX'd),N
DD 39	221 57	ADD IX,SP
DD 46 dd	221 70 dd	LD B,(IX'd)
DD 4E dd	221 78 dd	LD C,(IX'd)
DD 56 dd	221 86 dd	LD D,(IX'd)
DD 5E dd	221 94 dd	LD E,(IX'd)
DD 66 dd	221 102 dd	LD H,(IX'd)
DD 6E dd	221 110 dd	LD L,(IX'd)
DD 70 dd	221 112 dd	LD (IX'd),B

DD 71 dd	221 113 dd	LD (IX'd),C
DD 72 dd	221 114 dd	LD (IX'd),D
DD 73 dd	221 115 dd	LD (IX'd),E
DD 74 dd	221 116 dd	LD (IX'd),H
DD 75 dd	221 117 dd	LD (IX'd),L
DD 77 dd	221 119 dd	LD (IX'd),A
DD 7E dd	221 126 dd	LD A,(IX'd)
DD 86 dd	221 134 dd	ADD A,(IX'd)
DD 8E dd	221 142 dd	ADC A,(IX'd)
DD 96 dd	221 150 dd	SUB (IX'd)
DD 9E dd	221 158 dd	SBC A,(IX'd)
DD A6 dd	221 166 dd	AND (IX'd)
DD AE dd	221 174 dd	XOR (IX'd)
DD B6 dd	221 182 dd	OR (IX'd)
DD BE dd	221 190 dd	CP (IX'd)
DD E1	221 225	POP IX
DD E3	221 227	EX (SP),IX
DD E5	221 229	PUSH IX
DD E9	221 233	JP (IX)
DD F9	221 249	LD SP,IX
DD CB dd 06	221 203 dd 6	RLC (IX'd)
DD CB dd 0E	221 203 dd 14	RRC (IX'd)
DD CB dd 16	221 203 dd 20	RL (IX'd)
DD CB dd 1E	221 203 dd 30	RR (IX'd)
DD CB dd 26	221 203 dd 38	SLA (IX'd)
DD CB dd 2E	221 203 dd 46	SRA (IX'd)
DD CB dd 3E	221 203 dd 62	SRL (IX'd)
DD CB dd 46	221 203 dd 70	BIT 0,(IX'd)
DD CB dd 4E	221 203 dd 78	BIT 1,(IX'd)
DD CB dd 56	221 203 dd 86	BIT 2,(IX'd)
DD CB dd 5E	221 203 dd 94	BIT 3,(IX'd)
DD CB dd 66	221 203 dd 102	BIT 4,(IX'd)
DD CB dd 6E	221 203 dd 110	BIT 5,(IX'd)
DD CB dd 76	221 203 dd 118	BIT 6,(IX'd)
DD CB dd 7E	221 203 dd 126	BIT 7,(IX'd)
DD CB dd 86	221 203 dd 134	RES 0,(IX'd)
DD CB dd 8E	221 203 dd 142	RES 1,(IX'd)
DD CB dd 96	221 203 dd 150	RES 2,(IX'd)
DD CB dd 9E	221 203 dd 158	RES 3,(IX'd)
DD CB dd A6	221 203 dd 166	RES 4,(IX'd)
DD CB dd AE	221 203 dd 174	RES 5,(IX'd)
DD CB dd B6	221 203 dd 182	RES 6,(IX'd)
DD CB dd BE	221 203 dd 190	RES 7,(IX'd)
DD CB dd C6	221 203 dd 198	SET 0,(IX'd)

DD CB dd CE	221 203 dd 206	SET 1,(IX'd)
DD CB dd D6	221 203 dd 214	SET 2,(IX'd)
DD CB dd DE	221 203 dd 222	SET 3,(IX'd)
DD CB dd E6	221 203 dd 230	SET 4,(IX'd)
DD CB dd EE	221 203 dd 238	SET 5,(IX'd)
DD CB dd F6	221 203 dd 246	SET 6,(IX'd)
DD CB dd FE	221 203 dd 254	SET 7,(IX'd)
ED 40	237 64	IN B,(C)
ED 41	237 65	OUT (C),B
ED 42	237 66	SBC HL,BC
ED 43 XXXX	237 67 XXXX	LD (NN),BC
ED 44	237 68	NEG
ED 45	237 69	RETN
ED 46	237 70	IM 0
ED 47	237 71	LD I,A
ED 48	237 72	IN C,(C)
ED 49	237 73	OUT (C),C
ED 4A	237 74	ADC HL,BC
ED 4B XXXX	237 75 XXXX	LD BC,(NN)
ED 4D	237 77	RETI
ED 4F	237 79	LD R,A
ED 50	237 80	IN D,(C)
ED 51	237 81	OUT (C),D
ED 52	237 82	SBC HL,DE
ED 53 XXXX	237 83 XXXX	LD (NN),DE
ED 56	237 86	IM 1
ED 57	237 87	LD A,I
ED 58	237 88	IN E,(C)
ED 59	237 89	OUT (C),E
ED 5A	237 90	ADC HL,DE
ED 5B XXXX	237 91 XXXX	LD DE,(NN)
ED 5E	237 94	IM 2
ED 5F	237 95	LD A,R
ED 60	237 96	IN H,(C)
ED 61	237 97	OUT (C),H
ED 62	237 98	SBC HL,HL
ED 63 XXXX	237 99 XXXX	LD (NN),HL
ED 67	237 103	RRD
ED 68	237 104	IN L,(C)
ED 69	237 105	OUT (C),L
ED 6A	237 106	ADC HL,HL
ED 6B XXXX	237 107 XXXX	LD HL,(NN)
ED 6F	237 111	RLD
ED 72	237 114	SBC HL,SP

ED 73 XXXX	237 115 XXXX	LD (NN),SP
ED 78	237 120	IN A,(C)
ED 79	237 121	OUT (C),A
ED 7A	237 122	ADC HL,SP
ED 7B XXXX	237 123 XXXX	LD SP,(NN)
ED A0	237 160	LDI
ED A1	237 161	CPI
ED A2	237 162	INI
ED A3	237 163	OUTI
ED A8	237 168	LDD
ED A9	237 169	CPD
ED AA	237 170	IND
ED AB	237 171	OUTD
ED B0	237 176	LDIR
ED B1	237 177	CPIR
ED B2	237 178	INIR
ED B3	237 179	OTIR
ED B8	237 184	LDDR
ED B9	237 185	CPDR
ED BA	237 186	INDR
ED BB	237 187	OTDR
FD 09	253 9	ADD IY,BC
FD 19	253 25	ADD IY,DE
FD 21 XXXX	253 33 XXXX	LD IY,NN
FD 22 XXXX	253 34 XXXX	LD (NN),IY
FD 23	253 35	INC IY
FD 29	253 41	ADD IY,IY
FD 2A XXXX	253 42	LD IY,(NN)
FD 2B	253 43	DEC IY
FD 34 dd	253 52 dd	INC (IY'd)
FD 35 dd	253 53 dd	DEC (IY'd)
FD 36 dd XX	253 54 dd XX	LD (IY'd),N
FD 39	253 57	ADD IY,SP
FD 46 dd	253 70 dd	LD B,(IY'd)
FD 4E dd	253 78 dd	LD C,(IY'd)
FD 56 dd	253 86 dd	LD D,(IY'd)
FD 5E dd	253 94 dd	LD E,(IY'd)
FD 66 dd	253 102 dd	LD H,(IY'd)
FD 6E dd	253 110 dd	LD L,(IY'd)
FD 70 dd	253 112 dd	LD (IY'd),B
FD 71 dd	253 113 dd	LD (IY'd),C
FD 72 dd	253 114 dd	LD (IY'd),D
FD 73 dd	253 115 dd	LD (IY'd),E
FD 74 dd	253 116 dd	LD (IY'd),H

FD 75 dd	253 117 dd	LD (IY'd),L
FD 77 dd	253 119 dd	LD (IY'd),A
FD 7E dd	253 126 dd	LD A,(IY'd)
FD 86 dd	253 134 dd	ADD A,(IY'd)
FD 8E dd	253 142 dd	ADC A,(IY'd)
FD 96 dd	253 150 dd	SUB (IY'd)
FD 9E dd	253 158 dd	SBC A,(IY'd)
FD A6 dd	253 166 dd	AND (IY'd)
FD AE dd	253 174 dd	XOR (IY'd)
FD B6 dd	253 182 dd	OR (IY'd)
FD BE dd	253 190 dd	CP (IY'd)
FD E1	253 225	POP IY
FD E3	253 227	EX (SP),IY
FD E5	253 229	PUSH IY
FD E9	253 233	JP (IY)
FD F9	253 249	LD SP,IY
FD CB dd 06	253 203 dd 6	RLC (IY'd)
FD CB dd 0E	253 203 dd 14	RRC (IY'd)
FD CB dd 16	253 203 dd 20	RL (IY'd)
FD CB dd 1E	253 203 dd 30	RR (IY'd)
FD CB dd 26	253 203 dd 38	SLA (IY'd)
FD CB dd 2E	253 203 dd 46	SRA (IY'd)
FD CB dd 3E	253 203 dd 62	SRL (IY'd)
FD CB dd 46	253 203 dd 70	BIT 0,(IY'd)
FD CB dd 4E	253 203 dd 78	BIT 1,(IY'd)
FD CB dd 56	253 203 dd 86	BIT 2,(IY'd)
FD CB dd 5E	253 203 dd 94	BIT 3,(IY'd)
FD CB dd 66	253 203 dd 102	BIT 4,(IY'd)
FD CB dd 6E	253 203 dd 110	BIT 5,(IY'd)
FD CB dd 76	253 203 dd 118	BIT 6,(IY'd)
FD CB dd 7E	253 203 dd 126	BIT 7,(IY'd)
FD CB dd 86	253 203 dd 134	RES 0,(IY'd)
FD CB dd 8E	253 203 dd 142	RES 1,(IY'd)
FD CB dd 96	253 203 dd 150	RES 2,(IY'd)
FD CB dd 9E	253 203 dd 158	RES 3,(IY'd)
FD CB dd A6	253 203 dd 166	RES 4,(IY'd)
FD CB dd AE	253 203 dd 174	RES 5,(IY'd)
FD CB dd B6	253 203 dd 182	RES 6,(IY'd)
FD CB dd BE	253 203 dd 190	RES 7,(IY'd)
FD CB dd C6	253 203 dd 198	SET 0,(IY'd)
FD CB dd CE	253 203 dd 206	SET 1,(IY'd)
FD CB dd D6	253 203 dd 214	SET 2,(IY'd)
FD CB dd DE	253 203 dd 222	SET 3,(IY'd)

FD CB dd E6	253 203 dd 230	SET 4,(IY'd)
FD CB dd EE	253 203 dd 238	SET 5,(IY'd)
FD CB dd F6	253 203 dd 246	SET 6,(IY'd)
FD CB dd FE	253 203 dd 254	SET 7,(IY'd)

Appendix 3
Flag operation table

Flag table notation

Flags

–	Flag is unchanged by operation.
*	Flag is affected according to result of operation.
P	P/V is set according to parity result.
V	P/V is set according to the overflow result.
Ø	Flag is set to zero
1	Flag is set to one.
#	Result of flag unknown.
f	Contents of the interrupt flip flop.

Addressing

s	Any 8 bit addressing mode A, B, C, D, E, H, L, (HL), (IX+dd), (IY+dd)
r	Any 8 bit register A, C, D, E, H, L
b	Bit number 0–7
RR	Any 16 bit register.
n	Any 8 bit number

Flag operation table

Instruction		C	Z	P/V	S	N	H
ADC	HL,RR	*	*	V	*	Ø	#
ADC	A,s	*	*	V	*	Ø	*
ADC	A,n	*	*	V	*	Ø	*
ADD	A,s	*	*	V	*	Ø	*
ADD	A,n	*	*	V	*	Ø	*
ADD	HL,RR	*	–	–	–	Ø	#
ADD	SP,RR	*	–	–	–	Ø	#
ADD	IX,RR	*	–	–	–	Ø	#
ADD	IY,RR	*	–	–	–	Ø	#
AND	s	Ø	*	P	*	Ø	1
AND	n	Ø	*	P	*	Ø	1
BIT	b,s	–	*	#	#	Ø	1
CCF		*	–	–	–	Ø	#
CPD		–	*	*	#	1	#
CPDR		–	*	*	#	1	#
CPI		–	*	*	#	1	#
CPIR		–	*	*	#	1	#
CP	s	*	*	V	*	1	*
CP	n	*	*	V	*	1	*
CPL		–	–	–	–	1	1
DAA		*	*	P	*	–	*
DEC	s	–	*	V	*	1	*
IN	r,(C)	–	*	P	*	Ø	Ø
INC	s	–	*	V	*	Ø	*

Instruction		C	Z	P/V	S	N	H
IND		–	*	#	#	1	#
INI		–	*	#	#	1	#
INDR		–	1	#	#	1	#
INIR		–	1	#	#	1	#
LD	A,I	–	*	f	*	0	0
LD	A,R	–	*	f	*	0	0
LDD		–	#	*	#	0	0
LDI		–	#	*	#	0	0
LDDR		–	#	0	#	0	0
LDIR		–	#	0	#	0	0
NEG		*	*	V	*	1	*
OR	s	0	*	P	*	0	0
OR	n	0	*	P	*	0	0
OTDR		–	1	#	#	1	#
OTIR		–	1	#	#	1	#
OUTD		–	*	#	#	1	#
OUTI		–	*	#	#	1	#
RLA		*	–	–	–	0	0
RLCA		*	–	–	–	0	0

Instruction		C	Z	P/V	S	N	H
RRA		*	–	–	–	0	0
RRCA		*	–	–	–	0	0
RLD	(HL)	–	*	P	*	0	0
RRD	(HL)	–	*	P	*	0	0
RL	s	*	*	P	*	0	0
RLC	s	*	*	P	*	0	0
RR	s	*	*	P	*	0	0
RRC	s	*	*	P	*	0	0
SLA	s	*	*	P	*	0	0
SRA	s	*	*	P	*	0	0
SRL	s	*	*	P	*	0	0
SBC	HL,RR	*	*	V	*	1	#
SCF		1	–	–	–	0	0
SBC	A,s	*	*	V	*	1	*
SBC	A,n	*	*	V	*	1	*
SUB	s	*	*	V	*	1	*
SUB	n	*	*	V	*	1	*
XOR	s	0	*	P	*	0	0
XOR	n	0	*	P	*	0	0

Appendix 4
Spectrum monitor-assembler listing

```
        ASEG
        ORG     25500D

        JP      FSTART
ALTER   EQU     39D
COMMA   EQU     ','



SAVEB:  LD      A,0FFH
        ;IX POINTS TO START OF BLOCK
        ;DE CONTAINS  NUMBER OF BYTES
        CALL    04C2H
        RET

LOADB:  SCF
        LD      A,0FFH
        CALL    0556H
        RET

HEADIN:
        LD      DE,17
        LD      IX,HEADER
        XOR     A
        SCF
        CALL    0556H
        LD      A,(HEADER+11)
        LD      C,A
        LD      A,'$'
        LD      (HEADER+11),A
        CALL    CRLF

        LD      DE,HEADER+1
        CALL    PRTSTR
        LD      A,' '
        CALL    PRTCHR
```

```
        EX      DE,HL
        LD      (HL),C
        INC     HL
        INC     HL
        INC     HL

        LD      A,(HL)
        CALL    HEXO
        DEC     HL
        LD      A,(HL)
        CALL    HEXO
        DEC     HL
        LD      A,' '
        CALL    PRTCHR

        LD      A,(HL)
        CALL    HEXO
        DEC     HL
        LD      A,(HL)
        CALL    HEXO

        CALL    CRLF
        RET


HEADOUT:
        LD      DE,17
        LD      IX,HEADER
        XOR     A
        CALL    04C2H
        RET

KEY:    PUSH    HL
        PUSH    BC
        PUSH    DE
        CALL    WAITS           ;PAUSE FOR A WHILE
WAITK:  LD      A,(23611)       ;LOOK AT FLAGS
        BIT     5,A
        JR      Z,WAITK         ;NO KEY PRESSED
        RES     5,A             ;RESET FLAG
        LD      (23611),A
        POP     DE
```

```
        POP     BC
        POP     HL
        RET

GETKEY:
        CALL    KEY
        LD      A,(23560)       ;LOOK AT LAST-K
        CALL    PRTCHR
        RET
OPENCH2:
        LD      A,02            ;OPENS CHANNEL 'S'
                                ;FOR PRINTING
        CALL    1601H
        RET
PRTCHR:
        PUSH    AF
        PUSH    AF
        XOR     A
        LD      (23692),A
        POP     AF
        RST     10H
        POP     AF
        RET
PRTSTR:
        LD      A,(DE)          ;GET CHARACTER
        CP      '$'             ;IS THIS THE END
                                ;OF A STRING?
        RET     Z               ;YES THEN RETURN
        CALL    PRTCHR          ;PRINT CHARACTER
        INC     DE              ;POINT TO NEXT CHARACTER
        JR      PRTSTR
FSTART:
        LD      SP,STACK
        CALL    OPENCH2
        LD      DE,HOME
        CALL    PRTSTR
        LD      DE,WELCMESS
        CALL    PRTSTR
        CALL    CRLF

VERYSTART:
INITV2: LD      SP,STACK
        LD      A,8
        LD      (23658),A       ;CAPS ON
```

```
        LD      HL, ERRSP
        LD      (HL), LOW(VERYSTART)
        INC     HL
        LD      (HL), HIGH(VERYSTART)
        DEC     HL
        LD      (23613), HL

INITV:  LD      HL, INITV
        PUSH    HL

        CALL    CRLF
        LD      A,'>'
        CALL    PRTCHR

START:  CALL    GETKEY
        SUB     'A'             ;IS IT IN THE ALPHABET?
        RET     C               ;NO
        CP      'S'-'A'+1
        RET     NC              ;NO!
        ADD     A, A            ;*2
        LD      HL, VECTBL
        LD      E, A
        LD      D, 0
        ADD     HL, DE
        LD      E, (HL)
        INC     HL
        LD      D, (HL)
        EX      DE, HL
        JP      (HL)            ;JUMP TO COMMAND

VECTBL:
        DEFW    ERROR
        DEFW    ERROR
        DEFW    ERROR
        DEFW    DUMP            ;DUMP
        DEFW    MODIFY          ;EDIT MEMORY
        DEFW    FILL            ;FILL
        DEFW    GOTO            ;GOTO
        DEFW    HUNT            ;HUNT
        DEFW    IDENT           ;IDENTIFY FILENAME
        DEFW    ERROR
        DEFW    ERROR
```

```
        DEFW    LOADBYTES       ;LOAD FROM TAPE
        DEFW    MOVE            ;MOVE A BLOCK
        DEFW    ERROR
        DEFW    ERROR
        DEFW    PUMP            ;PRINT DUMP
        DEFW    ERROR
        DEFW    CHREG           ;MODIFY REGS
        DEFW    SAVEBYTES       ;SAVE MEMORY

WAITS:
        LD      BC,8000H
        LD      DE,4000H
        LD      HL,4000H
        LDIR
        RET


MODIFY:
        CALL    GETEXPR1        ;GET START ADDRESS
MODIFB: CALL    HEXOD           ;OUTPUT START ADDRESS
        LD      A,' '
        CALL    PRTCHR
        LD      A,(HL)
        CALL    HEXO
        LD      A,' '
        CALL    PRTCHR
        PUSH    HL
        CALL    HEXI
        POP     HL
        LD      (HL),A
        INC     HL
        CALL    CRLF
        JR      MODIFY          ;LOOP UNTIL FORCED
                                ;OUT BY AN ERROR




GETEXPR2:                       ;GET TWO WORD EXPRESSION
        CALL    HEXD
        PUSH    HL              ;SAVE E1
```

```
        LD      A,' '
        CALL    PRTCHR
        CALL    HEXD
        PUSH    HL
        POP     DE              ;E2
        POP     HL              ;GET E1
        RET                     ;E1=HL E2=DE

GETEXPR3:
        CALL    GETEXPR2
        PUSH    HL
        PUSH    DE
        CALL    GETEXPR1
        POP     DE
        POP     HL
        RET


GETEXPR1:

        LD      A,' '
        CALL    PRTCHR
        CALL    HEXD
        PUSH    HL
        POP     BC
        CALL    CRLF
        RET                     ;E1=HL E2=DE E3=BC


LOADBYTES:
        LD      A,' '
        CALL    PRTCHR
        CALL    GETEXPR2

        LD      (DESTT),HL

        LD      A,E
        OR      D
        JP      Z,ERRORS

        LD      (LENTT),DE

        LD      DE,LOADMESS
        CALL    PRTSTR
```

```
        LD      DE,FILENAME
        CALL    PRTSTR


GETFILE:
        CALL    HEADIN


        LD      DE,HEADER+1
        LD      HL,FILENAME
        LD      B,10

COMPF:  LD      A,(DE)
        RES     5,A
        LD      C,(HL)
        RES     5,C
        CP      C
        JR      NZ,GETFILE

        INC     HL
        INC     DE
        DJNZ    COMPF


LOAD1F: LD      DE,(LENTT)
        LD      IX,(DESTT)
        CALL    LOADB
        RET


DESTT:  DEFW    0
LENTT:  DEFW    0

CR      EQU     0DH             ;RETURN
LF      EQU     0AH             ;LINEFEED
WELCMESS: DEFM  CR,'*SBUG*'
          DEFM  ' (C) John Wilson'
          DEFM  ' 1984.'
          DEFM  CR,'$'
SAVEMESS: DEFM  CR,'Press any key when ready$'
```

```
LOADMESS: DEFM  CR,'Waiting for $'
NOTMESS:  DEFM  CR,'ROUTINE NOT IMPLEMENTED$'
ERRMESS:  DEFM  CR,'**ERROR**',CR,'$'
HOME:     DEFM  22,1,1,CR,'$'

SAVEBYTES:
          LD    A,' '
          CALL  PRTCHR
          CALL  GETEXPR2        ;GET 2 VALUES START
                                ;AND NUMBER OF BYTES
          LD    (HEADER+13),HL
          LD    A,E
          OR    D
          JP    Z,ERRORS
          LD    (HEADER+11),DE


          LD    DE,SAVEMESS
          CALL  PRTSTR

          CALL  WAITS
          CALL  WAITS
          CALL  WAITS
          CALL  GETKEY

          LD    A,3
          LD    DE,HEADER
          LD    HL,FILENAME
          LD    (DE),A
          INC   DE

          LD    BC,10
          LDIR
          CALL  HEADOUT
          CALL  WAITS
          CALL  WAITS
          CALL  WAITS
          LD    IX,(HEADER+13)
          LD    DE,(HEADER+11)
          CALL  SAVEB
          RET
```

```
HEXAS:  PUSH    HL
        CALL    HEXO
        LD      A,' '
        CALL    PRTCHR
        POP     HL
        RET

LINE:   LD      B,8
NBYTE:  LD      A,(HL)
        CALL    HEXAS
        INC     HL
        DJNZ    NBYTE
        CALL    CRLF
        RET


DUMP:
        LD      A,' '
        CALL    PRTCHR
        CALL    GETEXPR1
ALOCK:  LD      C,8
BLOCK:  CALL    HEXOD
        LD      A,' '
        CALL    PRTCHR
        CALL    PRTCHR
        CALL    LINE
        DEC     C
        JR      NZ,BLOCK
        CALL    CRLF
        CALL    CRLF
        CALL    GETKEY
        CP      CR
        JR      Z,ALOCK
        RET



PINE:   LD      B,21
PBYTE:  LD      A,(HL)
        CP      32
        JR      C,SBOGGY-2
        CP      128
        JR      C,SBOGGY
        LD      A,'.'
```

```
SBOGGY: CALL    PRTCHR
        INC     HL
        DJNZ    PBYTE
        CALL    CRLF
        RET


PUMP:
        LD      A,' '
        CALL    PRTCHR
        CALL    GETEXPR1
PALOCK: LD      C,8
PLOCK:  CALL    HEXOD
        LD      A,' '
        CALL    PRTCHR
        CALL    PRTCHR
        CALL    PINE
        DEC     C
        JR      NZ,PLOCK
        CALL    CRLF
        CALL    CRLF
        CALL    GETKEY
        CP      CR
        JR      Z,PALOCK
        RET




ERROR:  NOP
NOTIMP: PUSH    DE
        LD      DE,NOTMESS
        CALL    PRTSTR
        POP     DE
        RET

;ROUNTINES
;HEXO   OUTPUT HEX NUMBER IN ACCUMULATOR
;HEXOD  OUTPUT HEX WORD IN HL
;HEXI   INPUT HEX NUMBER PUT IN ACCUMULATOR
;HEXD   INPUT HEX WORD AND PUT INTO HL
```

```
HEXOD:
        LD      A,H
        CALL    HEXO
        LD      A,L

HEXO:
        LD      E,A
        SRL     A               ;GET TOP FOUR BITS
                                ;INTO LOWER NYBBLE
        SRL     A
        SRL     A
        SRL     A
        CALL    CONV            ;CONVERT TO ASCII
        ;RETURNS ASCII VALUE IN A
        LD      A,E             ;GET ORIGINAL VALUE
        AND     0FH             ;MASK OFF LOWER FOUR BITS
;CONVERT LAST HEX DIGIT
CONV:
        ADD     A,30H
        CP      3AH             ;IS DIGIT IN RANGE 0-9?
        JP      M,DECD          ;YES THEN PRINT AND RETURN
 ;IN THE RANGE 10-15 SO CONVERT TO A-F
        ADD     A,7
DECD:   CALL    PRTCHR          ;PRINT A HEX DIGIT
        RET

ERRORS: JP      VERYSTART

HEXI:
        CALL    GETKEY
        CALL    CONV2
        LD      E,A
        CALL    GETKEY
        CALL    CONV2
        SLA     E               ;MOVE LOWER FOUR BITS UP
        SLA     E
        SLA     E
        SLA     E
        OR      E               ;MERGE IN SECOND DIGIT
        RET
CONV2:  AND     A
        SBC     A,30H
        CP      0AH
```

```
        RET     C
        AND     A
        SBC     A,7
        CP      10H
        JR      NC,ERRORS
        RET

HEXD:   CALL    HEXI
        PUSH    AF
        CALL    HEXI
        LD      L,A
        POP     AF
        LD      H,A
        RET

FILL:   ;HL POINTS TO START ADDRESS
        ;DE POINTS END ADDRESS
        ;BC =NUMBER OF BYTES

        LD      A,' '
        CALL    PRTCHR
        CALL    GETEXPR2
        LD      A,' '
        CALL    PRTCHR


        PUSH    HL
        EX      DE,HL
        AND     A               ;CLEAR CARRY
        SBC     HL,DE
        JP      C,ERRORS
        JP      Z,ERRORS
        PUSH    HL
        POP     BC

        POP     HL
        PUSH    HL
        POP     DE
        INC     DE

        PUSH    HL
        PUSH    DE
        CALL    HEXI
```

```
        POP     DE
        POP     HL

        LD      (HL),A
        LDIR
        CALL    CRLF
        RET


RETGET: POP     DE              ;GET ORGINAL
        LD      HL,PUTREG
        PUSH    HL
        PUSH    DE
        CALL    GETREG
        RET

GOTO:



        LD      A,' '
        CALL    PRTCHR
        CALL    HEXD
        PUSH    HL              ;GOTO
        LD      A,' '
        CALL    PRTCHR

 ;POP    ALL REGS VALUES
        CALL    GETKEY
        CP      CR
        JR      Z,RETGET
        CP      COMMA
        JP      NZ,ERRORS
        CALL    HEXD
        PUSH    HL
        LD      DE,BRKP
        LD      BC,3
        LDIR                    ;SAVE BYTES

        POP     HL
        LD      (HL),0CDH
        INC     HL
        LD      (HL),LOW(BRK)
```

```
        INC     HL
        LD      (HL),HIGH(BRK)
        CALL    GETREG
        RET

PUTREG: LD      (SAVESP),SP
        LD      SP,AHLREG+2
        EX      AF,AF'
        EXX
        PUSH    HL
        PUSH    DE
        PUSH    BC
        PUSH    AF
        EXX
        EX      AF,AF'
        PUSH    IX
        PUSH    HL
        PUSH    DE
        PUSH    BC
        PUSH    AF
        LD      SP,(SAVESP)
        RET

GETREG: LD      (SAVESP),SP
        LD      SP,AFREG
        POP     AF
        POP     BC
        POP     DE
        POP     HL
        POP     IX
        EX      AF,AF'
        EXX
        POP     AF
        POP     BC
        POP     DE
        POP     HL
        EX      AF,AF'
        EXX
        LD      SP,(SAVESP)
        RET

BRK:    ;PUSH   ALL VALUES ON STACK
        CALL    PUTREG
```

```
        POP     HL              ;RET ADDRS
        DEC     HL
        DEC     HL
        DEC     HL
        ;BACK SPACE  3 INSTR

        CALL    CRLF
        LD      A,'*'
        CALL    PRTCHR
        CALL    HEXOD
        EX      DE,HL           ;DEST
        LD      HL,BRKP
        LD      BC,3
        LDIR                    ;PUT BYTES BACK

        ;PUT BACK FOUR BYTES
        ;DISPLAY PC
        ;DISPR:
        CALL    DISPR
        RET

OUTREG: LD      B,4
NXTREG: LD      E,(HL)          ;LOW
        INC     HL
        LD      D,(HL)
        PUSH    HL
        EX      DE,HL
DI1R:   CALL    HEXOD
        LD      A,' '
        CALL    PRTCHR
        CALL    PRTCHR
        POP     HL
        INC     HL
        DJNZ    NXTREG
        RET

DISPR:  CALL    CRLF
        LD      DE,REGMESS
        CALL    PRTSTR
        LD      HL,AFREG
        CALL    OUTREG
        CALL    IXOUT           ;DO IX REG
        CALL    CRLF            ;NOW ALTERNATE
```

```
        CALL    OUTREG
        RET

IXOUT:
DOING:  LD      A,(HL)
        PUSH    AF              ;SAVE LOW BYTE
        INC     HL
        LD      A,(HL)
        CALL    HEXO            ;OUT HIGH
        POP     AF              ;GET LOW
        CALL    HEXO            ;AND OUT
        INC     HL
        LD      A,' '
        CALL    PRTCHR
        CALL    PRTCHR
        RET



CHREG:
        ;GETREG VALUE

        LD      A,' '
        CALL    PRTCHR
        CALL    GETKEY
        CP      ALTER
        JR      NZ,LOO
        CALL    GETKEY

        ADD     A,'I'-'A'


LOO:    LD      HL,LOOKUP
        LD      BC,LENTAB
        CPIR
        JP      NZ,DISPR
        DEC     HL
        LD      DE,LOOKUP
        AND     A
        SBC     HL,DE
        LD      DE,AFREG
        SLA     L
        ADD     HL,DE
        INC     HL
```

```
        LD      A,' '
        CALL    PRTCHR
        LD      A,(HL)          ;LOW
        CALL    HEXO
        DEC     HL
        LD      A,(HL)          ;HIGH
        CALL    HEXO
        INC     HL
        LD      A,' '
        CALL    PRTCHR

        CALL    HEXI
        LD      (HL),A
        DEC     HL
        CALL    HEXI
        LD      (HL),A
        RET
LOOKUP:
        DB 'A','B','D','H','X'
        DB 'I','J','L','P'
   ;       A'  B'  D'  H'


LENTAB EQU      9D

MOVE:   LD      A,' '
        CALL    PRTCHR
        CALL    GETEXPR3

GETBC:  PUSH    HL
        AND     A
        SBC     HL,DE
        JR      NC,DSWOP
        POP     HL
        EX      DE,HL
        JR      GETBC
DSWOP:
        ;HL =NUMBER OF BYTES
        ;DE =START ADDRESS
        ;BC =DESTINATION
        ;(SP)=END ADDRESS
        EX      DE,HL
        ;DE =NUM HL=START
        ;BC=DEST
```

```
        PUSH    BC              ;SAVE DEST
        PUSH    DE              ;SAVE COUNT
        POP     BC              ;PUT IN DE
        POP     DE              ;GET DEST

        POP     AF              ;GET RID OF END
                                ;STACK CONTAINS  START
        PUSH    HL              ;GET START
                                ;HL CONTAINS START
                                ;DE DESTINATION
                                ;BC NUMBER OF BYTES
                                ;STACK CONTAINS
        AND     A
        SBC     HL,DE
        POP     HL
        JR      C,BACKW
        LDIR
        RET

BACKW:
        ADD     HL,BC
        DEC     HL
        EX      DE,HL
        ADD     HL,BC
        DEC     HL
        EX      DE,HL
        LDDR
        RET

IDENT:
        LD      A,' '
        CALL    PRTCHR
        CALL    GETKEY
        CP      0D
        RET     Z
        CP      65
        JP      C,ERRORS
        LD      HL,FILENAME
        LD      B,10
        LD      C,32
CLBUFF: LD      (HL),C
        INC     HL
        DJNZ    CLBUFF
```

```
        LD      HL,FILENAME
        LD      B,9

PUTBUF: LD      (HL),A
        DEC     B
        RET     Z

        INC     HL
        CALL    GETKEY

        CP      0DH
        RET     Z


        CP      65
        JP      C,ERRORS
        JR      PUTBUF





HUNT:
        LD      A,' '
        CALL    PRTCHR
        CALL    GETEXPR2
        PUSH    HL                  ;SAVE START
        EX      DE,HL
        AND     A
        SBC     HL,DE
        JP      C,ERRORS
        JP      Z,ERRORS
        PUSH    HL
        POP     BC
        POP     HL
        LD      A,' '
        CALL    PRTCHR

        PUSH    HL
        PUSH    DE
        CALL    HEXI
        POP     DE
        POP     HL
```

```
COMP:   CP      (HL)
        PUSH    AF
        JR      NZ,NFOUND
        CALL    CRLF
        CALL    HEXOD
        CALL    GETKEY
        CP      0DH
        JR      NZ,BHUN
NFOUND:
        INC     HL
        DEC     BC
        LD      A,B
        OR      C
        JR      Z,BHUN
        POP     AF
        JR      COMP
BHUN:
        POP     AF
        RET


CRLF:   LD      A,CR
        CALL    PRTCHR
        RET
REGMESS:
        DEFM    'AF     BC     '
        ;        0000::0000::
        DEFM    'DE     HL     '
        DEFM    'IX'
        DB      CR,'$'
REGS:
AFREG:  DEFW  0000H  ;AF
BCREG:  DEFW  0000H  ;BC
DEREG:  DEFW  0000H  ;DE
HLREG:  DEFW  0000H  ;HL
IXREG:  DEFW  0000H  ;IX
AAFREG: DEFW  0000H  ;AF'
ABCREG: DEFW  0000H  ;BC'
ADEREG: DEFW  0000H  ;DE'
AHLREG: DEFW  0000H  ;HL'

BRKP:   DB      0,0,0
SAVESP: DB      0,0
```

```
HEADER: DS      17
FILENAME:
        DS      10
        DB      CR,'$'

        DS      75
 STACK: DB      0
 ERRSP: DEFW    0

        END
```

Index

More Pan/PCN Computer Library titles for the Sinclair ZX Spectrum

Robert Erskine & Humphrey Walwyn,
Paul Stanley & Michael Bews
Sixty Programs for the Sinclair ZX Spectrum £5.95 0 330 28260 3

A massive software library for the price of a single cassette. Explosive games, dynamic graphics and invaluable utilities, this specially commissioned collection takes BASIC to the limits and beyond.

Four of the country's best-selling software writers have pooled their talents to bury programming clichés and exploit your micro's potential to the full.

Whether you are a games player or a more serious user, here's the book to make your micro work for you.

MASTER YOUR MICRO'S MOTHER TONGUE!

This practical guide to machine code programming lets the dedicated user harness the full power of the Spectrum's hardware and escape the confines of BASIC. The path to professional programming expertise is made clear as you are introduced to the Z80 instruction set and learn to combine the separate elements of machine code into the fast and efficient code found in commercial programs.

The annotated example programs allow you to enter and use fast screen handling routines and sorts in your own programs, debug them with the aid of the trace facility, run them in conjunction with the on-screen clock and lots more, while guiding you through the machine code maze (there's even a program to generate these) and illustrating the practical application of the techniques described.

The ROM routines, interrupt handling and programming principles are all covered, making this an essential reference and guide both for the beginner and the expert who's ready to branch out after succeeding in 'Cracking the Code'.

Cover photography by Peter Williams
with grateful acknowledgement for the kind assistance of
Chubb & Son's Lock & Safe Co. Ltd.

ISBN 0-330-28665-X

U.K. £6.95

90000

9 780330 286657